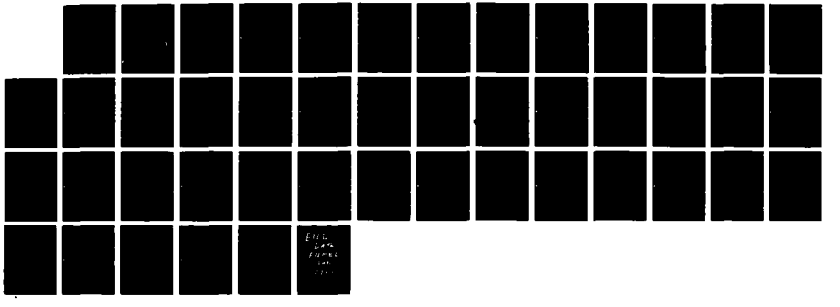
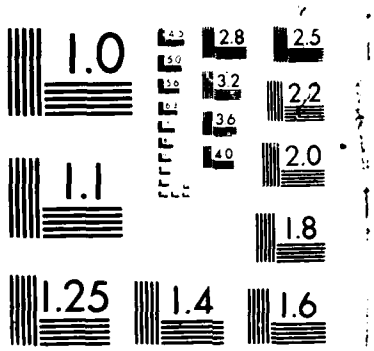
AD-A185 845 PROGRAM TRANSLATION VIA ABSTRACTION AND REIMPLEMENTATION(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB R C WATERS 1/1

UNCLASSIFIED DEC 86 AI-M949 N00014-85-K-0124 F/G 12/5 NL





(12)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER: AIM-949	2 GOVT ACCESSION NO	3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle): Program Translation Via Abstraction and Reimplementation		5 TYPE OF REPORT & PERIOD COVERED: AI-Memo
		6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s): Richard C. Waters		8 CONTRACT OR GRANT NUMBER(s): ONR: N00014-85-K-0124 (DARPA) NSF: MCS-7912179 IBM (no #) and SPERRY (no#)
9 PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139		10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209		12 REPORT DATE: December, 1986 13 NUMBER OF PAGES: 43
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217		15 SECURITY CLASS (of this report): UNCLASSIFIED 15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES

None

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program translation
Artificial Intelligence
Compilation
Programmer's Apprentice
Program Analysis

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

Essentially all program translators (both source-to-source and compilers) operate via transliteration and refinement. The source program is first transliterated into the target language on a statement by statement basis. Various refinements are then applied in order to improve the quality of the output. Although acceptable in many situations, this approach is fundamentally limited in the quality of output it can produce. In particular, it tends to be insufficiently sensitive to global features of the source program and too sensitive to irrelevant local details. [continued on reverse side]

(BLOCK #20 ABSTRACT-continued)

This paper presents an alternate translation paradigm - abstraction and reimplementation. Using this paradigm, the source program is first analyzed in order to obtain a programming language independent, abstract understanding of the computation performed by the program as a whole. The program is then reimplemented in the target language based on this understanding. The key to this appraoch is the abstract understanding obtained. It allows the translator to see the forest for the trees - benefiting from an appreciation of the global features of the source program without being distracted by irrelevant details.

Translation via abstraction and reimplementation is one of the goals of the Programmer's Apprentice project. A translator has been constructed which translates Cobol programs into Hibol (a very high level, business data processing language). A compiler has been designed which generates extremely efficient PDP-11 object code for Pascal programs. Currently, work is proceeding toward the implementation of a general purpose, knowledge-based translator.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 949 December 1986

Program Translation Via Abstraction and Reimplementation

by
Richard C. Waters

Abstract

Essentially all program translators (both source-to-source translators and compilers) operate via transliteration and refinement. The source program is first transliterated into the target language on a statement by statement basis. Various refinements are then applied in order to improve the quality of the output. Although acceptable in many situations, this approach is fundamentally limited in the quality of the output it can produce. In particular, it tends to be insufficiently sensitive to global features of the source program and too sensitive to irrelevant local details.

This paper presents an alternate translation paradigm—abstraction and reimplementation. Using this paradigm, the source program is first analyzed in order to obtain a programming language independent, abstract understanding of the computation performed by the program as a whole. The program is then reimplemented in the target language based on this understanding. The key to this approach is the abstract understanding obtained. It allows the translator to see the forest for the trees—benefiting from an appreciation of the global features of the source program without being distracted by irrelevant details.

Translation via abstraction and reimplementation is one of the goals of the Programmer's Apprentice project. A translator has been constructed which translates Cobol programs into Hibol (a very high level, business data processing language). A compiler has been designed which generates extremely efficient PDP-11 object code for Pascal programs. Currently, work is proceeding toward the implementation of a general purpose, knowledge-based translator.

(To appear in *IEEE Transactions on Software Engineering.*)

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the IBM Corporation, in part by the Sperry Corporation, in part by National Science Foundation grant MCS-7912179, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, either expressed or implied, of the IBM Corporation, of the Sperry Corporation, of the National Science Foundation, or of the Department of Defense.

1 - Introduction

The goal of this paper is to present the idea of translation via abstraction and reimplementation and compare it with the standard approach of translation via transliteration and refinement. In the main, this is done through a discussion of the basic ideas behind the two approaches and a discussion of the designs for two translators based on abstraction and reimplementation. In addition, the paper presents a detailed description of an implemented prototype translator which demonstrates the efficacy of the abstraction and reimplementation approach.

The process of program translation takes a program written in some source language and creates an equivalent program in some target language. The primary goal of translation is to create a syntactically correct program in the target language which computes the same thing as the source program in more or less the same way. For a wide variety of source and target languages, satisfying this goal is relatively straightforward.

In addition to the primary goal of correctness, translation typically has one or more subsidiary goals such as efficiency or readability of the target program. In general, the most difficult aspect of translation is not producing correct output, but rather attempting to satisfy these subsidiary goals. The main problem is that typically the subsidiary goals of translation are at best orthogonal to, and at worst in conflict with, the goals of the original author of the source program.

Translations vary widely in quality. An optimal translation would produce the program which the original authors would have produced had they been writing in the target language in the first place and had they had the desired subsidiary goals in mind.

The most common example of program translation is compilation — the translation of a program written in a high level language into machine language. In compilation, the key subsidiary goal is achieving efficiency in the target program. The work on compilers has demonstrated that acceptable efficiency can be obtained. However, there is still a long way to go. Even the best optimizing compilers fall short of the efficiency which programmers can achieve writing directly in machine language.

Another important application of program translation is source-to-source program translation. In this situation, a program is translated from a language which may be in some way obsolete into another language where it can be more easily maintained. In source-to-source translation, the key subsidiary goal is achieving readability (and hence maintainability) of the target program. The use of automatic translation during maintenance has been severely limited by the fact that readability of the target program is very difficult to achieve.

Most current program translators operate by a process which could be called translation via *transliteration and refinement*. In this process, the source program is first transliterated into the target language on a line by line basis by translating each line in isolation. Various refinements are then applied in order to improve the target program produced. As discussed in Section II, this process has a number of advantages. However, it is inherently limited in the extent to which it can satisfy the subsidiary goals of translation. In particular, translation via transliteration and refinement tends to be insufficiently sensitive to global features of the source program and too sensitive to irrelevant local details of the source program.

Section III presents an alternative approach to program translation — translation via *abstraction and*

reimplementation. In this process, the source program is first analyzed in order to obtain an abstract description of the computation being performed. The program is then reimplemented in the target language based on the abstract description. The central feature of this approach is the abstraction step. It allows the translator to benefit from a global understanding of what the source program does. In addition, the abstraction step deliberately discards information about details of the source program which are not relevant to the translation process. Although inherently more complex than translation via transliteration and refinement, translation via abstraction and reimplementation is capable of producing very high quality results.

Sections IV & V present examples of program translators which operate via abstraction and reimplementation. The first example translator (Satch [10]) is a prototype system which translates Cobol programs into Hibol. (Hibol is a very high level, non-procedural, business data processing language.) Satch is notable because it produces extremely readable output. The second example (Cobbler [9]) is a proposed compiler which translates Pascal programs into PDP-11 assembler language. Cobbler is notable because it produces extremely efficient output.

Section VI describes efforts within the Programmer's Apprentice project [28] toward the construction of a general purpose, knowledge-based translation system operating via abstraction and reimplementation. In order to support very high quality translation, this system will have extensive knowledge of how algorithms can be expressed in the source and target languages. In order to make the system general purpose, this knowledge will be represented declaratively in a library of algorithm schemas. Each schema will specify how a class of algorithms can be rendered in the source or target language.

Section VII discusses other work which is relevant to the idea of translation via abstraction and reimplementation. In particular, research on natural language translation has shown that obtaining a global understanding of the source text is essential for producing high quality translations.

II - Translation via Transliteration and Refinement

As shown in Fig. 1, translation via transliteration and refinement operates in two steps. The transliteration step translates the source program on an element by element basis. (The word *transliteration* (as opposed to *translation*) is used to connote the idea of literal translation where each element is translated in isolation without regard for context.) The output of the transliteration step is expressed either directly in the target language or in an intermediate language which is semantically similar to it.

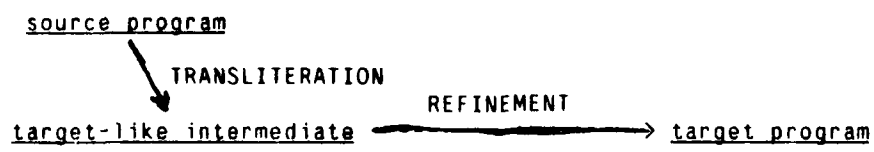


Fig. 1. Translation via transliteration and refinement.

The refinement step takes the output of the transliteration step and applies various correctness preserving transformations in order to improve its quality. For example, compilers apply optimizations in order to improve

the efficiency of the code produced. If the intermediate language is not identical to the target language, then the refinement step also performs the (typically trivial) translation from intermediate to final form.

Example of Transliteration and Refinement

As an example of translation via transliteration and refinement, consider how this approach could be used to translate Fortran [34] programs into Ada [37] programs. Fig. 2 shows a Fortran program BOUND which is taken from the IBM Fortran Scientific Subroutine Package [35]. Fig. 3 shows the result of the transliteration step of the translation process. Fig. 4 shows the final result after the refinement step of the translation process.

The program BOUND has six input parameters and four output parameters. The parameter A is a matrix which contains a set of observations of a number of variables presumably determined in some experiment. The integer parameters NO and NV specify the number of observations and the number of variables respectively. (As is generally the case in the programs in the Scientific Subroutine Package, although A is logically a matrix, it is declared to be a vector and all of the index computations are explicit in the program.)

The parameter S is a vector of length NO. The vector S selects the observations which should be considered by the program BOUND. An observation J is considered only if S(J) is non-zero.

The parameters BLO and BHI are vectors of length NV. For each variable, these vectors specify lower and upper bounds respectively for the observation values. The integer parameter IER is used to return an error code. If BLO(I)>BHI(I) for any I then IER is set to one and computation is aborted; otherwise it is set to zero.

The parameters UNDER, BETW, and OVER are also vectors of length NV. For each variable I, the program BOUND counts how many of the selected observations are under BLO(I), how many are between BLO(I) and BHI(I) inclusive, and how many are over BHI(I). These counts are stored in the variables UNDER, BETW, and OVER respectively which are the principal outputs of the program BOUND.

```
      SUBROUTINE BOUND(A,S,BLO,BHI,UNDER,BETW,OVER,NO,NV,IER)
      DIMENSION A(1),S(1),BLO(1),BHI(1),UNDER(1),BETW(1),OVER(1)
      IER = 0
      DO 10 I = 1, NV
      IF (BLO(I)-BHI(I)) 10,10,11
   11 IER = 1
      GO TO 12
   10 CONTINUE
      DO 1 K = 1, NV
      UNDER(K) = 0.0
      BETW(K) = 0.0
    1 OVER(K) = 0.0
      DO 8 J = 1, NO
      IJ = J-NO
      IF (S(J)) 2,8,2
    2 DO 7 I = 1, NV
      IJ = IJ+NO
      IF (A(IJ)-BLO(I)) 5,3,3
    3 IF (A(IJ)-BHI(I)) 4,4,6
    4 BETW(I) = BETW(I)+1.0
      GO TO 7
    5 UNDER(I) = UNDER(I)+1.0
      GO TO 7
    6 OVER(I) = OVER(I)+1.0
    7 CONTINUE
    8 CONTINUE
   12 RETURN
      END
```

Fig. 2. The Fortran program BOUND.

The transliteration process is illustrated by Fig. 3. Each part of the program is translated locally. The Fortran parameters are all turned into "`in out`" parameters of appropriate types in the Ada program. They are given the mode "`in out`" because every Fortran parameter can potentially be both an input value and an output value. The Fortran assignment statements are converted into equivalent Ada assignments. This requires very little change because Fortran is essentially a subset of Ada when it comes to arithmetic expressions and assignment statements. Fortran arithmetic IFs are expanded into equivalent Ada "`if then else`" statements branching to the appropriate labels. Arithmetic IFs where two of the labels are the same are treated as special cases in order to avoid the need for temporary variables. Each Fortran DO is expanded into an equivalent Ada "`loop`". The Ada "`for`" construct cannot be used because Ada "`for`" tests for termination at the top of the loop while Fortran DO tests for termination at the bottom of the loop. Fortran CONTINUE, RETURN and GO TO are turned into Ada "`null`", "`return`", and "`goto`" respectively. The only aspect of the transliteration which is not totally local is that the Fortran program has to be scanned in order to determine what variables are used in the program so that appropriate variable declarations can be inserted at the beginning of the Ada program.

```
type VECTOR is array (INTEGER range <>) of REAL;
procedure BOUND(A,S,BLO,BHI,UNDER,BETW,OVER: in out VECTOR;
                NO,NV,IER: in out INTEGER) is
        I,IJ,J,K: INTEGER;
begin
        IER := 0;
        I := 1;
        loop;
           if BLO(I)-BHI(I)<=0.0 then goto L10;
           else goto L11;
           end if;
   <<L11>> IER := 1;
           goto L12;
   <<L10>> null;
           I := I+1;
           exit when I>NV;
        end loop;
        K := 1;
        loop
           UNDER(K) := 0.0;
           BETW(K) := 0.0;
    <<L1>> OVER(K) := 0.0;
           K := K+1;
           exit when K>NV;
        end loop;
        J := 1;
        loop
           IJ := J-NO;
           if S(J)=0.0 then goto L8;
           else goto L2;
           end if;
    <<L2>> I := 1;
           loop
              IJ := IJ+NO;
              if A(IJ)-BLO(I)<0.0 then goto L5;
              else goto L3;
              end if;
       <<L3>> if A(IJ)-BHI(I)<=0.0 then goto L4;
              else goto L6;
              end if;
       <<L4>> BETW(I) := BETW(I)+1.0;
              goto L7;
       <<L5>> UNDER(I) := UNDER(I)+1.0;
              goto L7;
       <<L6>> OVER(I) := OVER(I)+1.0;
       <<L7>> null;
              I := I+1;
              exit when I>NV;
           end loop;
    <<L8>> null;
           J := J+1;
           exit when J>NO;
        end loop;
<<L12>> return;
end BOUND;
```

Fig. 3. A transliteration of Fig 2 into Ada.

As is typically the case with transliteration, the program in Fig. 3, although correct, does not do a good job of satisfying the subsidiary goals of translation (in this case readability). Fig. 4 shows the final result after the refinement step of the translation process.

```
type VECTOR is array (INTEGER range <>) of REAL;
procedure BOUND(A,S,BLO,BHI: VECTOR;
                UNDER,BETW,OVER: in out VECTOR;
                NO,NV: INTEGER; IER: out INTEGER) is
     I,IJ,J,K: INTEGER;
begin
     IER := 0;
     I := 1;
     loop
        if BLO(I)-BHI(I)<=0.0 then goto L10; end if;
        IER := 1;
        return;
<<L10>> I := I+1;
        exit when I>NV;
     end loop;
     K := 1;
     loop;
        UNDER(K) := 0.0;
        BETW(K) := 0.0;
        OVER(K) := 0.0;
        K := K+1;
        exit when K>NV;
     end loop;
     J := 1
     loop;
        IJ := J-NO;
        if S(J)=0.0 then goto L8; end if;
        I := 1;
        loop;
           IJ := IJ+NO;
           if A(IJ)-BLO(I)<0.0 then goto L5; end if;
           if A(IJ)-BHI(I)>0.0 then goto L6; end if;
           BETW(I) := BETW(I)+1.0;
           goto L7;
    <<L5>> UNDER(I) := UNDER(I)+1.0;
           goto L7;
    <<L6>> OVER(I) := OVER(I)+1.0;
    <<L7>> I := I+1;
           exit when I>NV;
        end loop;
 <<L8>> J := J+1;
        exit when J>NO;
     end loop;
end BOUND;
```

Fig. 4. A refined transliteration of Fig 2 into Ada.

Fig. 4 is derived from Fig. 3 by applying a number of correctness-preserving transformations. Complex "`if then else`" statements which have clauses which branch to the next statement are simplified to remove these clauses. The branch to a "`return`" statement is replaced by a "`return`" statement. Unnecessary "`null`" statements, "`return`" statements, and labels are removed. Instead of giving all the parameters the mode "in out", some of the parameters are given just the mode "out" or "in" (the default in Ada). This is done in a

purely syntactic way by noting that parameters which are never assigned to cannot be "out" and parameters which are never read cannot be "in".

There are a number of transformations which could in principle have been applied to the program which have not been. For example, the computation involving UNDER, BETW, and OVER could be rearrange into one large "if then else". However, in keeping with the kinds of refinements typically supported by source-to-source translators (see Section VII), two criteria were used in order to decide which refinements to perform. First, no support was provided for transformations which require either control flow or data flow analysis of the program. This rules out transformations like the one suggested above.

Second, the main emphasis was placed on transformations which only look at an adjacent pair of statements. The only transformation which is more complicated than this is the one which refines the mode of the parameters. This transformation has to scan the program in order to determine which parameters are read and assigned. However, it does not do an actual data flow analysis. If it did, it would realize that UNDER, BETW, and OVER are actually "out" parameters and not "in out" parameters since they cannot be read until after they have been assigned.

Fig. 4 is readable, but still not as good as one would like. In particular, it falls far short of the goal of producing the program the programmers would have produced had they been writing in Ada — it is a Fortran-style Ada program instead of an Ada-style Ada program. As will be discussed in Section III, better translations of Fig. 2 can be achieved by means of translation via abstraction and reimplementation.

Figs. 3 & 4 are not the output of any particular translator. Rather, they are hypothetical examples intended to illustrate the process of transliteration and refinement. However, it is not clear that any existing source-to-source translator produces output which is significantly better than Fig. 4 (see Section VII).

Advantages of Transliteration and Refinement

Translation via transliteration and refinement has several advantages. Most importantly, it uses a divide and conquer strategy in order to satisfy the goals of translation. The basic goal of obtaining a correct translation is achieved by the transliteration step. The refinement step need only guarantee that it preserves this correctness. The subsidiary goals of the translation (e.g., efficiency or readability) are achieved by the refinement step. The transliteration step is greatly simplified by not having to worry about the subsidiary goals.

Another advantage is that the localized nature of the transliteration step makes it easy to encode the basic knowledge needed for translation. This knowledge is economically represented by stating how each of the constructs in the source language should be converted into equivalent constructs in the target language. The transliteration step need not have any knowledge about how special combinations of source constructs can be represented as special combinations of target constructs. (This latter kind of knowledge is the province of the refinement step which presumably knows how to fine tune cumbersome combinations of target constructs.)

A final advantage of translation via transliteration and refinement is that it makes it easy to construct families of [illegible] translators which share the same transliteration step or share the same refinement step. For example, one

might construct a family of compilers which compile various high level languages into the same machine language and which share the same refinement step.

Transliteration Is Not Always Practical

Although it works satisfactorily in many situations, translation via transliteration and refinement has some fundamental disadvantages. To begin with, it assumes that transliteration is practical. This in turn depends on the assumption that each of the source language constructs can be individually translated into target language constructs in a practical way. Unfortunately, this is not always the case.

The main way in which transliteration can be blocked is that the source language may support a primitive construct which is not supported by the target language. For example, consider translating from a language which supports `GOTO`s into a language which does not, or from a language which supports multiple assignments to a variable into a functional language which does not. In the case of Fortran and Ada, consider the fact that Ada has nothing which is equivalent to the Fortran `EQUIVALENCE` statement.

The primary source of incompleteness in current translators is primitive constructs which cannot be transliterated. Current translators typically just ignore non-transliteratable constructs, either refusing to process source programs which contain them or copying them unchanged from the source to the target. Human intervention is required either to remove them from the source or to fix them up in the target.

A second way in which transliteration can be blocked is that the source and target languages may have constructs which, although they correspond closely, differ in significant semantic details. Most of the time these details may not matter for translation. However, when they matter they are liable to matter a lot. For example, consider translating into a language which forces complex data structures to be copied when they are assigned to a variable from a language which does not, or between languages which differ in their variable scoping rules. In the case of Fortran and Ada, consider the fact that vector arguments to Fortran subroutines are passed by reference while Ada specifies that it is undefined whether or not vector arguments will be copied or passed by reference.

The primary source of incorrectness in current translators is constructs which can be transliterated straightforwardly most of the time but only with great difficultly (or not at all) in certain hard-to-detect situations. Current translators typically just use the straightforward transliteration all of the time without giving any indication that there might be a problem. (For example, the transliteration in Fig. 3 blindly assumes that it does not matter how the vector parameters get passed.) Human intervention is required in order to correct any problems which arise in the target program produced.

Transliteration Complicates Refinement

A second fundamental disadvantage of translation via transliteration and refinement is an unintended byproduct of its greatest advantage. The principal virtue of the transliteration and refinement approach is that it simplifies the problem of satisfying the primary goal of translation (i.e., correctness) by factoring out the problem of satisfying the subsidiary goals of translation. Unfortunately, this factoring typically complicates the task of

satisfying the subsidiary goals. This is particularly unfortunate since the subsidiary goals are usually harder to satisfy than the primary goal.

The basic reason why translation via transliteration and refinement complicates the task of satisfying the subsidiary goals of translation is that typically the process of transliteration does not merely ignore the subsidiary goals, it works against them. Simply put, whether or not the original source program is *good* from the point of view of the subsidiary goals of the translation, the output of the transliteration step is almost always guaranteed to be *bad* from this point of view.

The most obvious way in which transliteration makes things difficult for later refinement is that, more often than not, the transliteration of a given construct in the source language requires the use of a circumlocution in the target language. The only time when this can be completely avoided is when the target language possess a semantically identical construct. Examples of both of these cases can be seen in Fig. 3. The `DO` loops in the Fortran program are converted into cumbersome "`loop`" statements in the Ada program. In contrast, the assignment statements remain essentially unchanged.

A more subtle way in which transliteration makes things difficult for later refinement is that it tends to obscure the key features of the algorithm implemented by the program being translated. Transliteration does this through both camouflage and the creation of decoys. The mass of circumlocutions produced by transliteration act as camouflage hiding the key features. Decoys (features which are prominent but actually unimportant) are created because the code produced is sensitive to unimportant details of the source. For example, Fig. 3 would have looked quite different if the Fortran programmer had used logical `IF`s instead of arithmetic `IF`s. A kind of indirect camouflage is produced due to the fact that the transliteration step is insensitive to global considerations. Transliteration typically renders a given construct in exactly the same way even if the context would suggest that it should be translated differently. For example, all of the parameters are given the mode "`in out`" in Fig. 3 whether or not this is actually necessary given the way they are used.

A final way in which transliteration makes things difficult for later refinement is that useful information about the source program can get lost. As an example of this, consider translating from a language (such as Ada) where the order of evaluation of the arguments of a function call is undefined to a language where the order is defined. In this situation, straightforward transliteration will define an evaluation order and thereby discard the information that many evaluation orders are equally acceptable. This loss of information makes it hard for the refinement step to apply transformations which are not applicable to the chosen evaluation order but which are applicable to one of the evaluation orders which was not chosen.

Applicability of Transliteration and Refinement

The primary requirement for the applicability of translation via transliteration and refinement is that transliteration must be practical. For this to be the case, the target language must support all of the primitive constructs supported by the source language. In general, this implies that the target language must be at a lower level than the source language — i.e., have a wider variety of primitive constructs. This is why the arrow in Fig. 1

between the source and the intermediate language is drawn pointing downward.

Translation via transliteration and refinement is perhaps most applicable to compilation because any primitive construct can be expressed in machine language. In contrast, source-to-source translators typically have to restrict the input language and/or admit possibly incorrect translations in order to make transliteration practical.

A second limitation on the applicability of translation via transliteration and refinement is that refinement is an inherently difficult task which transliteration makes more difficult. As a result, the transliteration and refinement approach is most applicable in situations where the subsidiary goals of translation are not too stringent.

Transliteration and refinement works well in a straightforward compiler where readability of the output is not an issue and only moderate efficiency is required in the output code. In order to achieve significantly higher levels of efficiency in the output code, optimizing compilers expend an enormous amount of effort on refinement.

III - Translation via Abstraction and Reimplementation

As shown in Fig. 5, translation via abstraction and reimplementation operates in two steps. The abstraction step performs a global analysis of the source program. The goal of this analysis is to obtain an understanding of the algorithms being used by the program. The abstract description highlights the essential features of these algorithms while deliberately throwing away information about unimportant features of the program.

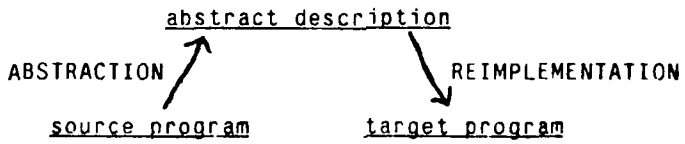


Fig. 5. Translation via abstraction and reimplementation.

The reimplementation step takes the abstract description produced by the abstraction step and creates a program in the target language which implements this description. In order to simplify this task, the abstract description is designed so that it contains exactly the right kind of information needed in order to guide the reimplementation process.

The basic difference between translation via transliteration and refinement and translation via abstraction and reimplementation can be seen by comparing the shapes of Figs. 1 & 5. The transliteration and refinement approach translates directly to the target language. In contrast, the abstraction and reimplementation approach first translates the source program up to a very high level description and then translates this description down to the target language.

Like translation via transliteration and refinement, translation via abstraction and reimplementation uses a divide and conquer strategy to attack the translation task. However, it divides the translation task differently. The transliteration and refinement approach separates the problem of satisfying the primary goal of translation from the problem of satisfying the subsidiary goals of translation. In contrast, the abstraction and reimplementation approach separates knowledge of the source language from knowledge of the target language.

Example of Abstraction and Reimplementation

As an example of translation via abstraction and reimplementation, consider how this approach could be used to translate the Fortran program in Fig. 2 into Ada. The first step is to obtain an abstract description of the computation in Fig. 2. Fig. 6 shows the key elements of such an abstract description.

Fig. 6 is divided into three parts. The first part lists the parameters of the program BOUND and their types as specified in the original Fortran program. (By convention, the Scientific Subroutine Package uses the dimension specification V(1) to specify a vector of unknown length rather than a vector of length one.) A complete data flow analysis of the program is used in order to determine which parameters are "in" and which are "out". This analysis reveals that UNDER, BETW, and OVER are never read before they are written and are therefore "out" parameters.

The second part of Fig. 6 lists a number of constraints which must be satisfied in order for the program BOUND to produce reasonable results. The first seven constraints state that the ranges of the various vector parameters must be large enough to prevent referencing memory locations outside of the vectors. These constraints are determined by looking at the largest values which the various index variables in the program can reach.

The last two constraints specify that the parameters NO and NV must be positive and therefore that the vector parameters must have positive extent. These are particularly interesting constraints because they imply that Ada "for" loops can be used when translating the program. The constraints follow from the observation that a Fortran DO loop which enumerates the elements of an array does not operate correctly when given an array of zero extent. The problem is that the body of a Fortran DO loop is always executed at least once, even if the limits placed on the DO variable suggest that zero executions would be more appropriate. (This feature of DO is occasionally used in a constructive way by Fortran DO loops which do not enumerate the elements of arrays.)

The third part of Fig. 6 describes the computation performed by the program. The first two lines specify that the program checks to see that every element of BLO is less than or equal to the corresponding element of BHI. If this is true then IER is set to zero. Otherwise, IER is set to one and the program is terminated.

The remainder of Fig. 6 describes the main computation performed by the program BOUND in terms of recurrence equations. The main body of the program is a doubly nested loop iterating over the index variables J and I. The various evaluations of the body of the inner loop can be referred to in terms of the corresponding values of the index variables. The notation $X_{m,n}$ is used to refer to the value of the variable X at the end of the evaluation of the inner loop body during which the outer loop index has the value m and the inner loop index has the value n. The recurrence equations specify how variable values corresponding to a given evaluation of the inner loop body are computed from values corresponding to earlier evaluations. The recurrence equations are derived by inspecting the data flow in the loops. As part of this process, the middle loop in the Fortran code is revealed to be part of the initialization for the main loop in the program.

The fact that Fig. 6 is shown in a textual form is not intended to imply that the abstract description would actually be represented textually. For example, it might take the form of logical expressions annotating a data flow graph representing the program.

```
PARAMETERS:
  in A,S,BLO,BHI: vector of real
 out UNDER,BETW,OVER: vector of real
  in NO,NV: integer
 out IER: integer
CONSTRAINTS:
  A'RANGE⊇1..NV*NO,  S'RANGE⊇1..NO,
  BLO'RANGE⊇1..NV,  BHI'RANGE⊇1..NV,
  UNDER'RANGE⊇1..NV,  BETW'RANGE⊇1..NV,  OVER'RANGE⊇1..NV,
  NO≥1,  NV≥1
COMPUTATION:
  if (∀ I∈1..NV BLO(I)≤BHI(I)) then IER=0
     else IER=1 ∧ computation is aborted
  The main computation is a doubly nested loop
    The outer index (first subscript) counts from 1 to NO
    The inner index (second subscript) counts from 1 to NV
    The variables assigned within the loops have the following values:
      ∀ j∈1..NO, i∈1..NV, K∈1..NV
        IJ_{j,0}=j-NO
        IJ_{j,i}=IJ_{j,i-1}+NO
        UNDER(K)_{0,i}=0.0
        if K=i ∧ S(j)≠0.0 ∧ A(IJ_{j,i})<BLO(i)
           then UNDER(K)_{j,i}=1.0+UNDER(K)_{j-1,i}
           else UNDER(K)_{j,i}=UNDER(K)_{j-1,i}
        BETW(K)_{0,i}=0.0
        if K=i ∧ S(j)≠0.0 ∧ BLO(i)≤A(IJ_{j,i})≤BHI(i)
           then BETW(K)_{j,i}=1.0+BETW(K)_{j-1,i}
           else BETW(K)_{j,i}=BETW(K)_{j-1,i}
        OVER(K)_{0,i}=0.0
        if K=i ∧ S(j)≠0.0 ∧ BHI(i)<A(IJ_{j,i})
           then OVER(K)_{j,i}=1.0+OVER(K)_{j-1,i}
           else OVER(K)_{j,i}=OVER(K)_{j-1,i}
```

Fig. 6. An abstract description of Fig. 2.

Based on the abstract description in Fig. 6, it is a straightforward matter to create a quality translation of the program BOUND into Ada as shown in Fig. 7. The parameters are made parameters in the code with the specified types. The recurrence equations map directly into a triply nested loop. Transformations similar to those used by an optimizing compiler can be used to get rid of the unnecessary innermost loop over K and to move the test S(J)/=0.0 out to the outermost loop since it is an invariant in the inner loop.

A comparison of Fig. 7 with Fig. 4 shows that the translation in Fig. 7 is superior in several respects. Most notably, the parameters have all been given the correct modes; labels and "goto" statements have been eliminated in favor of complex "if then else" statements; and "for" loops have been used.

Some of the improvements which are seen in Fig. 7 could have been achieved in Fig. 4 if local refinement had been applied more aggressively. For example, local transformations probably could have been used to combine the simple "if then else" statements in Fig. 4 with the statements following them in order to create the

"if then else" statements shown in Fig. 7.

However, improvements such as determining the proper modes for the parameters and utilizing "for" loops depend critically on an understanding of the program as a whole. These changes cannot be made until after a global analysis of the program has determined that the changes are valid.

```
type VECTOR is array (INTEGER range <>) of REAL;
procedure BOUND(A,S,BLO,BHI: VECTOR;
                UNDER,BETW,OVER: out VECTOR;
                NO,NV: INTEGER; IER: out INTEGER) is
    I,IJ,J,K: INTEGER;
begin
    IER := 0;
    for I in 1..NV loop
        if BLO(I)>BHI(I) then IER := 1; return; end if;
    end loop;
    for K in 1..NV loop
        UNDER(K) := 0.0;
        BETW(K) := 0.0;
        OVER(K) := 0.0;
    end loop;
    for J in 1..NO loop
        if S(J)/=0.0 then
           IJ := J-NO;
           for I in 1..NV loop
               IJ := IJ+NO;
               if A(IJ)<BLO(I) then UNDER(I) := UNDER(I)+1.0;
               elsif BHI(I)<A(IJ) then OVER(I) := OVER(I)+1.0;
               else BETW(I) := BETW(I)+1.0;
               end if;
           end loop;
        end if;
    end loop;
end BOUND;
```

Fig. 7. A translation of Fig 2 into Ada based on Fig. 6.

While Fig. 7 is a good translation of Fig. 2 into Ada, it is still far from optimal. Appropriate Ada-style constructs have been used, however, the result is still essentially a Fortran-style program. In particular, the fact that A is really a matrix, but is declared to be a vector and the fact that the various vector parameters may have ranges which are larger than the ranges indicated by the parameters NO and NV is in the style of the Fortran Scientific Subroutine Package, but, it is not in the style of Ada.

Fig. 7 is shown as it is because it is just about the best translation which can be achieved if the parameters and their types are required to remain the same as in the Fortran program. In addition, it illustrates the kind of translation which can be achieved by using an abstract representation which is only moderately abstract.

Example of Increased Abstraction

Figs. 8 & 9 show a translation of the program BOUND into Ada which is better than the one shown in Fig. 7 and the abstract description on which it is based. There are two fundamental ways in which the translation shown in these figures is different from the one shown in Figs. 6 & 7. First, Figs. 8 & 9 assume that the program BOUND and

the programs which call it are being translated together. This opens up two new avenues of attack on the translation problem. The programs which call BOUND can be inspected in order to obtain additional information about BOUND. The interface to the program BOUND can be altered in order to render the program more aesthetically in Ada.

In Fig. 8 it is assumed that an analysis of the programs which call BOUND shows that BOUND is only called with vectors which have the exact sizes indicated by the parameters NO and NV. This makes it possible to tighten up the constraints in the description and to eliminate all mention of the variables NO and NV in favor of using the Ada array attribute "'RANGE" applied to the parameters.

The second fundamental difference between Figs. 8 & 9 and Figs. 6 & 7 is that Fig. 8 is significantly more abstract than Fig. 6. The computations being performed are described in terms of their net effects. The computations involving UNDER, BETW, and OVER are described as computing a count of elements of A which have certain properties. The variable S is described as a vector of flags which are tested. A is described directly as a matrix, and no mention is made of the variable IJ. The computation involving IER is summarized by stating that the computation is aborted and an error signalled if the first constraint is violated. No mention is made of how this might be done.

```
LOGICAL INPUTS:
  A matrix of real
  S vector of flag
  BLO,BHI vector of real
LOGICAL OUTPUTS:
  UNDER,BETW,OVER vector of count
  error signaled (and computation aborted) if constraint (1) is violated
CONSTRAINTS:
 (1) ∀ I∈BLO'RANGE BLO(I)≤BHI(I)
 (2) A'RANGE(1)=BLO'RANGE=BHI'RANGE=UNDER'RANGE=BETW'RANGE=OVER'RANGE
 (3) A'RANGE(2)=S'RANGE
COMPUTATION:
  ∀ I∈UNDER'RANGE
     UNDER(I) = count-of {J∈S'RANGE | S(J) ∧ A(I,J)<BLO(I)}
  ∀ I∈BETW'RANGE
     BETW(I) = count-of {J∈S'RANGE | S(J) ∧ BLO(I)≤A(I,J)≤BHI(I)}
  ∀ I∈OVER'RANGE
     OVER(I) = count-of {J∈S'RANGE | S(J) ∧ BHI(I)<A(I,J)}
```

Fig. 8. A more abstract description of Fig. 2.

The key to the increase in abstraction in Fig. 8 is the ability to *recognize* the net effects of a computation. This in turn depends on the abstraction component having a significant amount of knowledge about what kinds of computations can be performed. For example, it can presumably recognize that the recurrence equations in Fig. 6 compute counts and that the computation involving the variable IJ converts matrix indices to vector indices. Similarly, it can recognize that the computation involving the variable IER reflects the standard way that error conditions are signalled in the Fortran Scientific Subroutine Library.

Based on Fig. 8, the reimplementation step can produce a much better program (see Fig. 9) than the one

shown in Fig. 7 because it has fewer restrictions placed on it. It can choose better parameters and better types because the abstract description does not require that the parameters and types be the same as in the Fortran program. It is free to implement the error signalling using standard Ada methods — i.e., by raising an exception instead of returning an error value which has to be explicitly checked by the caller. Due to the stronger constraints on the length of the vectors, array literals can be used to initialize the vectors UNDER, BETW, and OVER instead of a loop.

In some situations, the added freedom does not cause any change in the translation. For example, the reimplementation step could have computed the counts in several different ways. However, none of these methods would have been any better than the one shown in Fig. 7, so the same method was used in Fig. 9.

There is a price which has to be payed in order to get the improved translation shown in Fig. 9. Analysis is made more complicated by the need to recognize the net effects of the computation being performed. In addition, reimplementation is made more complicated because there are more implementation decisions which have to be made.

```
type VECTOR is array (INTEGER range <>) of REAL;
type BOOLS is array (INTEGER range <>) of BOOLEAN;
type VECT is array (INTEGER range <>) of INTEGER;
type MATRIX is array (INTEGER range <>, INTEGER range <>) of REAL;
procedure BOUND(A: MATRIX; S: BOOLS; BLO,BHI: VECTOR;
                UNDER,BETW,OVER: out VECT) is
    I,J: INTEGER;
begin
    for I in BLO'RANGE loop
        if BLO(I)>BHI(I) then raise CONSTRAINT_ERROR; end if;
    end loop;
    UNDER := (UNDER'RANGE => 0);
    BETW := (BETW'RANGE => 0);
    OVER := (OVER'RANGE => 0);
    for J in A'RANGE(2) loop
        if S(J) then
           for I in A'RANGE(1) loop
               if A(I,J)<BLO(I) then UNDER(I) := UNDER(I)+1;
               elsif BHI(I)<A(I,J) then OVER(I) := OVER(I)+1;
               else BETW(I) := BETW(I)+1;
               end if;
           end loop;
        end if;
    end loop;
end BOUND;
```

Fig. 9. A translation of Fig 2 into Ada based on Fig. 8.

Figs. 6-9 are not produced by any particular translator. Rather, they are hypothetical examples intended to illustrate the process of abstraction and reimplementation. In particular, they demonstrate that increased abstraction leads to improved translation. In the limit, it is possible to create a translation which compares favorably with the program the programmers would have written had they been writing in the target language.

Advantages of Abstraction and Reimplementation

The most important advantage of translation via abstraction and reimplementation is that, while translation via transliteration and refinement is, in essence, designed to facilitate achieving the primary goal of translation (i.e., correctness), translation via abstraction and reimplementation is specifically designed to facilitate achieving the subsidiary goals of translation. As discussed in Section II, transliteration creates many problems for later refinement. In contrast, the sole purpose of abstraction is to simplify later reimplementation. Sections IV & V give extended examples of the way in which abstraction and reimplementation can cooperate in order to produce high quality translation.

A second important advantage of translation via abstraction and reimplementation is that it is not limited by the practicality of transliteration. As discussed in Section II, the local nature of transliteration can cause it to be blocked even though overall translation is possible. In contrast, there is no *a priori* reason for abstraction to ever be blocked since the result of abstraction is not constrained by the target language. Further, reimplementation need not be blocked as long as overall translation is possible.

A final virtue of translation via abstraction and reimplementation is that it lends itself to the construction of families of translators which share components at least as well as translation via transliteration and refinement if not better. In this regard, note that designing an abstract representation which is compatible with a diverse set of target languages is easier then designing a target-like intermediate language which is compatible with them.

Disadvantages of Abstraction and Reimplementation

Like translation via transliteration and refinement, translation via abstraction and reimplementation has a fundamental problem of incompleteness. Unlike transliteration, abstraction and reimplementation are always possible as long as translation is possible. However, it would not be reasonable to assume that these processes will always be practical. When they are not, a translator will have to fall back on some other method of translation. For example, it might use transliteration (or ask for human assistance) in order to translate those parts of a program which could not be usefully abstracted and/or reimplemented.

A key issue then is what percentage of a typical source program can be practically abstracted and reimplemented. This question can only be answered in the context of a particular application. However, two general statements can be made. First, any particular deficiency in abstraction or reimplementation can be rectified by adding more knowledge into the abstraction and reimplementation modules. Second, the limits of abstraction and reimplementation are essentially orthogonal to the limits of transliteration. Therefore, a translator which uses abstraction and reimplementation and which falls back on transliteration should always be more complete than one which uses transliteration alone.

Another disadvantage of the abstraction and reimplementation approach is that it is more complicated than transliteration and refinement. All in all, in situations where transliteration is practical and little refinement is necessary, translation via transliteration and refinement is probably the approach of choice. However, in situations where transliteration is not practical or where [illegible] its subsidiary goals,

translation via abstraction and reimplementation can succeed in producing high quality output where translation via transliteration and refinement would fail.

IV - Satch — Translating from Cobol to Hibol

Faust's Satch system [10] uses abstraction and reimplementation in order to attack a problem which is particularly difficult for translation by transliteration and refinement — translation from a low level programming language to a high level programming language. There are two key problems with this kind of translation. First, transliteration is usually not practical. Second, the subsidiary goal of such a translation is readability which is an exceptionally difficult goal to satisfy well.

In the case of Satch, the source language is Cobol [36] and the target language is Hibol [20]. The motivation behind the translation performed by Satch is the desire to convert pre-existing Cobol programs into a form where they can be more easily maintained. The benefits of the translation are illustrated by the fact that the resulting Hibol program can be as much as an order of magnitude shorter than the original Cobol program.

Hibol is a special purpose business data processing language. It is a very high level, non-procedural, single assignment language which is based on the concept of a *flow*. A flow is a multidimensional aggregate of data values which are indexed by one or more keys. Each Hibol statement specifies how a flow is computed from other flows. This is done by specifying how a typical element of the output flow is computed from typical elements of the input flows. An important advantage of Hibol is that both file I/O and iteration over the elements of flows is implicit in a Hibol program and therefore does not have to be explicitly specified by the programmer. Fig. 11 (which will be discussed below) shows an example of a Hibol program.

A key aspect of the non-procedural nature of Hibol is that there is no explicit control flow in a Hibol program. The statements in a Hibol program are unordered and there are no flow of control constructs such as conditionals or loops. As a result of this, direct transliteration from a programming language such as Cobol which has flow of control constructs to Hibol is not practical.

Example of Satch's Translation

Figs. 10 & 11 (adapted from [10]) show an example of a translation performed by Satch. Fig. 10 shows a Cobol program named PAYROLL. This program reads in a file of records which specify the wage rate for each member of a group of employees. The program computes the gross pay for each employee based on a 40 hour week along with a count of the employees and the total gross pay for all the employees.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT HOURLY-WAGE-IN ASSIGN TO DA-2301-S-HWI.
   SELECT GROSS-PAY-OUT ASSIGN TO DA-2301-S-GPO.
   SELECT EMPLOYEE-COUNT-OUT ASSIGN TO DA-2301-S-ECO.
   SELECT TOTAL-GROSS-PAY-OUT ASSIGN TO DA-2301-S-TGPO.
DATA DIVISION.
FILE SECTION.
FD hourly-wage-in
   LABEL RECORD IS OMITTED
   DATA RECORD IS hourly-wage-rec.
01 hourly-wage-rec.
   02 employee-number                    PICTURE IS 9(9).
   02 hourly-wage                        PICTURE IS 999V99.
FD gross-pay-out
   LABEL RECORD IS OMITTED
   DATA RECORD IS gross-pay-rec.
01 gross-pay-rec.
   02 employee-number                    PICTURE IS 9(9).
   02 gross-pay                          PICTURE IS 999V99.
FD employee-count-out
   LABEL RECORD IS OMITTED
   DATA RECORD IS employee-count-rec.
01 employee-count-rec.
   02 employee-count                     PICTURE IS 9(6).
FD total-gross-pay-out
   LABEL RECORD IS OMITTED
   DATA RECORD IS total-gross-pay-rec.
01 total-gross-pay-rec.
   02 total-gross-pay                    PICTURE IS 9(7)V99.
PROCEDURE DIVISION.
initialization SECTION.
        MOVE ZERO TO total-gross-pay.
        MOVE ZERO TO employee-count.
        OPEN INPUT hourly-wage-in.
        OPEN OUTPUT gross-pay-out.
mainline SECTION.
        READ hourly-wage-in AT END GO TO end-of-job.
        MOVE employee-number OF hourly-wage-rec
          TO employee-number OF gross-pay-rec.
        MULTIPLY hourly-wage BY 40 GIVING gross-pay.
        ADD 1 TO employee-count.
        ADD gross-pay TO total-gross-pay.
        WRITE gross-pay-rec.
        GO TO mainline.
end-of-job SECTION.
        CLOSE hourly-wage-in.
        CLOSE gross-pay-out.
        OPEN OUTPUT employee-count-out.
        WRITE employee-count-rec.
        CLOSE employee-count-out.
        OPEN OUTPUT total-gross-pay-out.
        WRITE total-gross-pay-rec.
        CLOSE total-gross-pay-out.
        STOP RUN.
```

Fig. 10. The Cobol program PAYROLL.

Fig. 11 shows the Hibol translation which is produced by Satch. Like any Hibol program, this program is divided into two parts which are closely analogous to the parts of a Cobol program. The data division of the Hibol program specifies the data types of the flows (introduced by the keyword FILE) used in the program and how these flows are indexed. The computation division specifies how the output flows are computed from the input flows. The first line of the computation division specifies that the elements of the flow GROSS-PAY are computed by multiplying the elements of the flow HOURLY-WAGE by 40. The second line of the computation division specifies how to compute the single element flow TOTAL-GROSS-PAY. The operator SUM collapses a dimension of a flow by adding all of the elements in that dimension together. In an analogous way, the third line of the computation division specifies how to count the number of employees.

```
DATA DIVISION
   KEY SECTION
      KEY EMPLOYEE-NUMBER  FIELD TYPE IS NUMBER  FIELD LENGTH IS 9
   INPUT SECTION
      FILE HOURLY-WAGE  KEY IS EMPLOYEE-NUMBER
   OUTPUT SECTION
      FILE GROSS-PAY  KEY IS EMPLOYEE-NUMBER
      FILE EMPLOYEE-COUNT
      FILE TOTAL-GROSS-PAY
COMPUTATION DIVISION
   GROSS-PAY IS (HOURLY-WAGE * 40.)
   TOTAL-GROSS-PAY IS (SUM OF (HOURLY-WAGE * 40.))
   EMPLOYEE-COUNT IS (COUNT OF HOURLY-WAGE)
```

Fig. 11. Satch's translation of Fig. 10 into Hibol.

Without discussing Figs. 10 & 11 in any more detail, it can be seen that Satch is capable of creating quite good Hibol translations of Cobol programs. (More complex examples are given in [10].) However, the translations produced by Satch are still not optimal. For example, it would be better if Satch were capable of realizing that the flow TOTAL-GROSS-PAY in Fig. 11 could be computed using the more compact expression (SUM OF GROSS-PAY).

Implementation of Satch

Like the architecture of any translation system based on abstraction and reimplementation, Satch's architecture is divided into two basic parts (see Fig. 12). The five modules on the left side of the figure operate together to create an abstract description of the Cobol program supplied to Satch. The Hibol reimplementation module creates a Hibol program based on the abstract description. Most of the burden of the translation is carried by the abstraction modules. This asymmetry is due to the fact that the very high level nature of Hibol allows the abstract description to be similar to the target language.

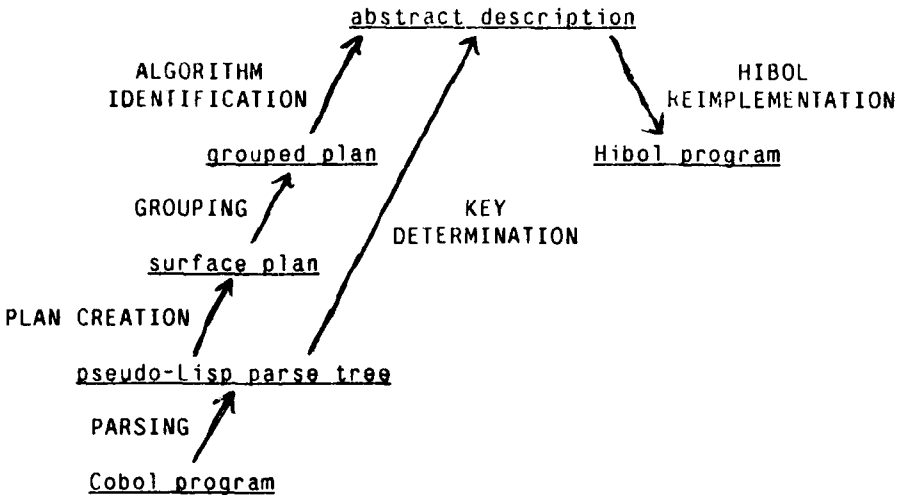


Fig. 12. The architecture of Satch.

The parsing module (implemented by G. Burke) parses the Cobol program and transliterates it into pseudo-Lisp. (Lisp [24] was chosen as the output of this module in order to facilitate the use of a pre-existing plan creation module.) The parsing module is implemented in essentially the same way that the transliteration component of a Cobol to Lisp translator operating via transliteration and refinement would be implemented.

For each file in the Cobol program, the key determination module determines which of the fields of the file act as keys. Various heuristics could be used to determine this information by looking at the Cobol program. However, Satch currently asks the user to specify which fields are key fields. In ordinary use this would not lead to an excessive amount of user interaction because key determination only has to be done once for each file even if a large number of programs which operate on the files are being translated.

The plan creation module converts the pseudo-Lisp output of the parsing module into a programming language independent internal representation called a *surface plan*. Fig. 13 (adapted from [10]) shows a simplified version of the surface plan which Satch creates when operating on the Cobol program PAYROLL shown in Fig. 10.

A plan is similar to a data flow diagram. Computations are represented by boxes (called segments). The segments are connected by solid arrows indicating data flow and dashed arrows indicating control flow. In the figure, many of the data flow arrows have annotations indicating the variables they correspond to. The names of the segments represent the operations they perform. PLUS adds two numbers. CREAD reads a record from a file. EOFP determines whether the end of a file has been reached. PIF splits control flow based on whether or not its input is TRUE.

In the interest of brevity, the plan in Fig. 13 has been simplified in several ways. The computation of EMPLOYEE-COUNT has been omitted. The file open and close functions have been removed. Except for the file HOURLY-WAGE, the data flow corresponding to the various file objects has been omitted. The data flow for the file HOURLY-WAGE was retained in order to make the EOFP test understandable.

Plan creation is performed using global data flow and control flow analysis which is similar to the kind of analysis which is performed by an optimizing compiler.

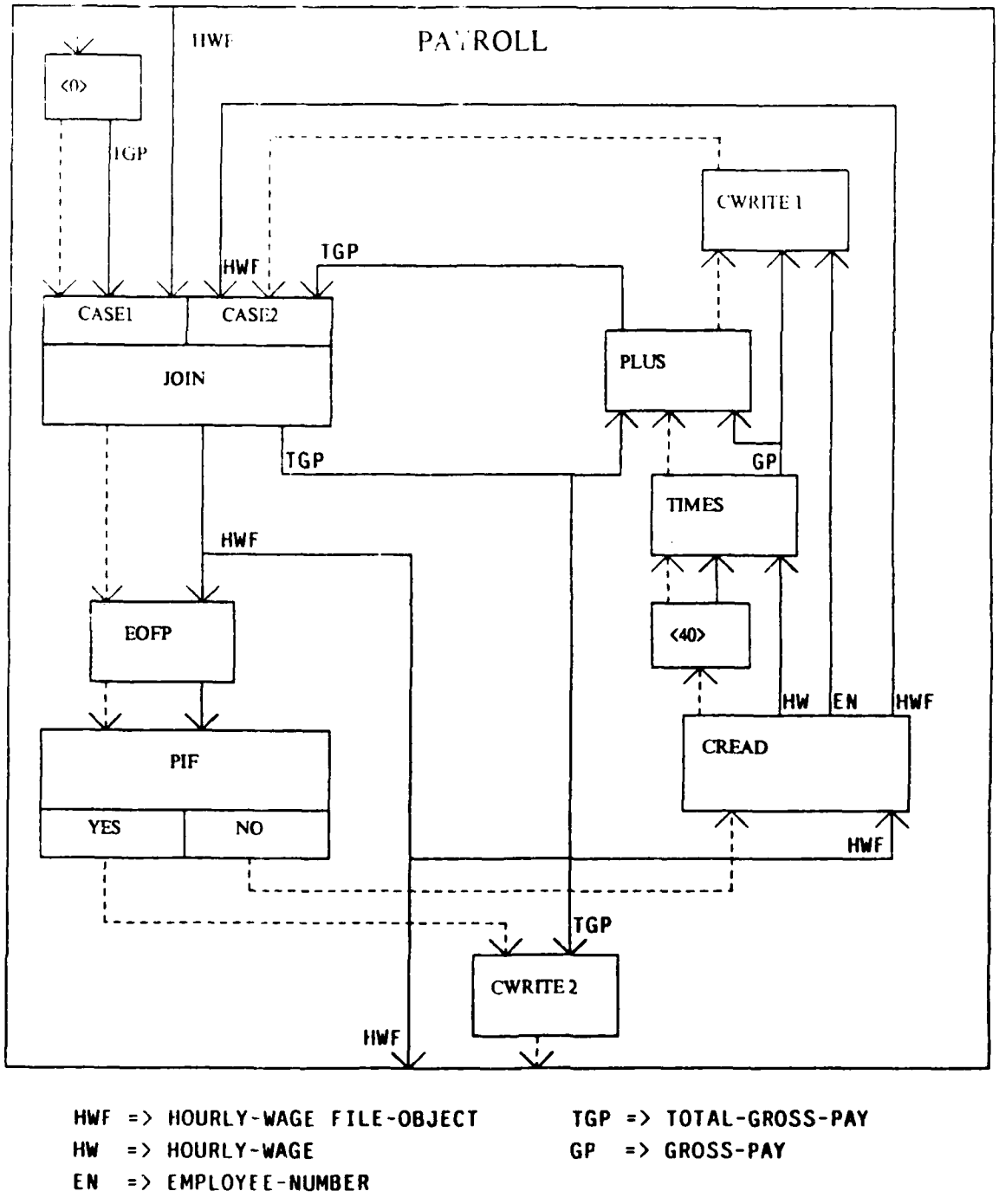


Fig. 13. A simplified surface plan for PAYROLL.

The grouping module takes the surface plan generated by the plan creation module and converts it into a *grouped* plan. A grouped plan differs from a surface plan in two ways. First, the segments in the plan are grouped

into a hierarchy of segments within segments in order to highlight the logical structure of the plan. Second, the loops in the plan are identified and broken down into their component parts.

Fig. 14 shows a simplified grouped plan for the program PAYROLL. Like Fig. 13 the grouped plan omits the file open and close functions and some of the other file operations. The figure is also simplified in that it does not show the computation which occurs within the various segments. Unlike Fig. 13 the grouped plan shows the computation of EMPLOYEE-COUNT.

The key difference between Figs. 13 & 14 is the way the loop in the program PAYROLL is represented. In Fig. 14, the various parts of the loop are broken apart into segments which are connect by data flow rather than control flow. This is done through a process called *temporal abstraction* [27].

Temporal abstraction treats series of values in the loop (e.g., the successive values of HOURLY-WAGE) as if they were single data objects. These *temporal series* are represented by bold data flow arrows in Fig. 14. Temporal abstraction analyzes a loop as a set of *generators* and *consumers* which are sources and sinks for temporal series. For example, in Fig. 14, the generator CREAD creates a temporal series of HOURLY-WAGE values which are consumed by the segment TIMES(40). This segment in turn creates a temporal series of GROSS-PAY values which are summed up by the segment PLUS(SUM).

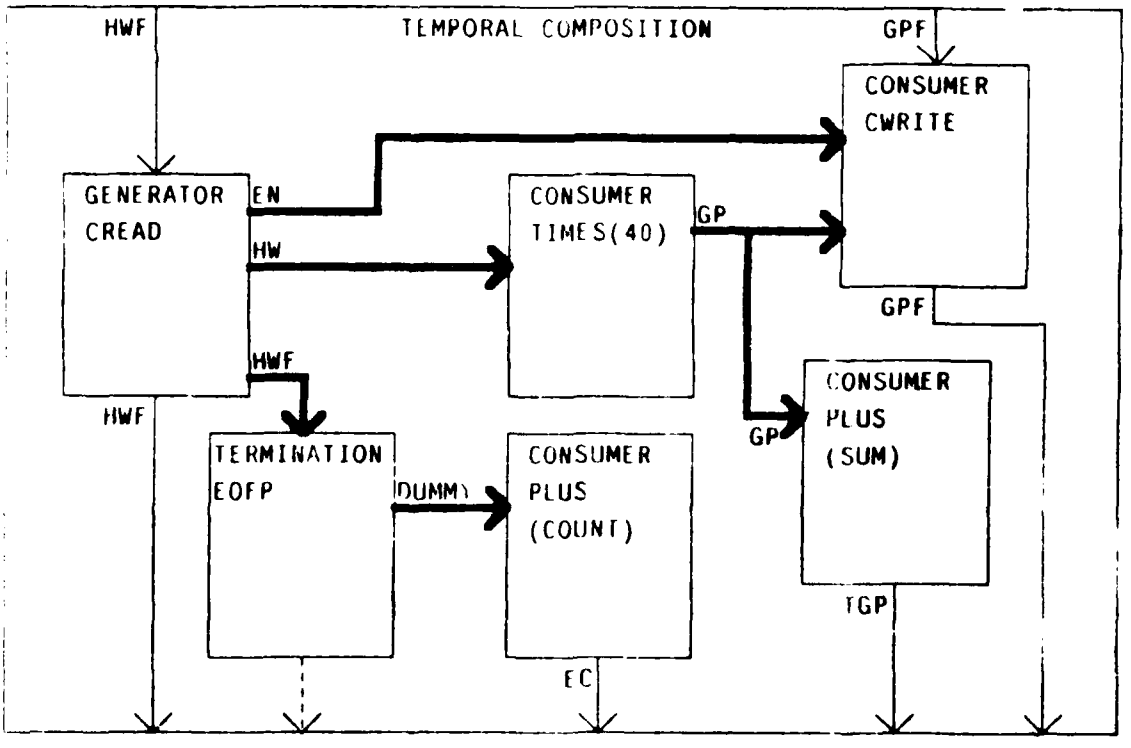


Fig. 14. A simplified grouped plan for PAYROLL.

As discussed in detail in [27] the process of temporal abstraction is based on the data flow in a loop. [illegible] consumer nodes [illegible] the loop, which [illegible]

understood in isolation.

Satch was implemented in the context of the Programmer's Apprentice project and it shares many ideas with the rest of the project. In particular, the plan representation, the plan creation module, and the grouping module are borrowed directly from KBEmacs [28] which is the current demonstration system developed as part of the Programmer's Apprentice project.

The algorithm identification module inspects the grouped plan and determines the net effect of the computation being performed. In combination with the results of key determination, the results of algorithm identification form an abstract description of the program. Fig. 15 (adapted from [10]) shows the abstract description which is created for the program PAYROLL. The first part of Fig. 15 comes directly from the data division of the Cobol program annotated by the key determination module. The second part of Fig. 15 comes from algorithm identification.

Algorithm identification operates in two stages. The first stage identifies what kinds of looping computations are present in the program. This is done by special purpose procedures which scan the grouped plan and recognize standard kinds of computation. In Fig. 14, these recognition procedures identify that the segments CREAD and EOFP enumerate the records in a file while the segment CWRIT... accumulates a series of records into a file. They also identify that the segment PLUS(SUM) computes a sum while the segment PLUS(COUNT) computes a count. (The names of these segments in Fig. 14 reflect the fact that this recognition has been performed.) The recognition stage of the algorithm identification module makes it possible to use the terms "enumerate", "sum", and "count" in the abstract description to describe the computation in the loop instead of recurrence equations.

The second stage of algorithm identification computes summary descriptions of the computation performed by the program. This is done by means of a symbolic evaluator which traverses the plan and accumulates algebraic equations which describe the computation. For example, the symbolic evaluator determines that the field GROSS-PAY has the value "CREAD-VALUE(HOURLY-WAGE-IN, HOURLY-WAGE)*40." — i.e., forty times the value of the HOURLY-WAGE field read from the file HOURLY-WAGE-IN. Similarly, it determines that the field TOTAL-GROSS-PAY accumulates the sum of the GROSS-PAY values. An algebraic simplifier is used in order to render the equations in as compact a form as possible.

```
FILES:
  HOURLY-WAGE-IN
     key-field EMPLOYEE-NUMBER-IN 9(9)
    data-field HOURLY-WAGE 999V99
  GROSS-PAY-OUT
     key-field EMPLOYEE-NUMBER-OUT 9(9)
    data-field GROSS-PAY 999V99
  EMPLOYEE-COUNT-OUT
    data-field EMPLOYEE-COUNT 9(6)
  TOTAL-GROSS-PAY-OUT
    data-field TOTAL-GROSS-PAY 9(7)V99
COMPUTATION:
  The main loop in the program enumerates the records in the file
    HOURLY-WAGE-IN.  It terminates when EOFP(HOURLY-WAGE-IN).
  fields written on each cycle of the main loop:
    EMPLOYEE-NUMBER-OUT = CREAD-VALUE(HOURLY-WAGE-IN, EMPLOYEE-NUMBER-IN)
    GROSS-PAY = CREAD-VALUE(HOURLY-WAGE-IN, HOURLY-WAGE)*40.
  fields written after the main loop:
    EMPLOYEE-COUNT = count(NOT(EOFP(HOURLY-WAGE-IN)))
    TOTAL-GROSS-PAY = sum(CREAD-VALUE(HOURLY-WAGE-IN, HOURLY-WAGE)*40.)
```

Fig. 15. An abstract description of PAYROLL.

The reimplementation module of Satch produces a Hibol program based on the abstract description of the Cobol program. This is done by converting these equations into Hibol syntax. The only real complexity in this is checking that the program is expressible in Hibol. In particular, the reimplementation module has to check that each input file is processed in full and that the input keys map to the output keys in a way which is compatible with the implicit file reading and writing performed by Hibol.

Limits of Satch

Although it illustrates the efficacy of translation based on abstraction and reimplementation, there are several ways in which Satch is limited. First of all, Satch is only a demonstration system. It has only been tested on a few examples and therefore has not been fully debugged. In addition, it is quite slow.

A more fundamental problem with Satch is that it is only applicable to a narrow class of Cobol programs. Part of this is due to the fact that, since Hibol is a relatively special purpose language, many Cobol programs cannot be reasonably translated into Hibol by any means. However, there are many Cobol programs which could in principle be translated into Hibol in a reasonable way which cannot be translated by Satch. The basic difficulty is that Satch does not have a generalized recognition facility. Rather, special purpose procedures have to be written in order for Satch to be able to identify what kinds of looping computations are present in a program. Overcoming this difficulty is a primary goal of the knowledge-based translation system discussed in Section VI.

V - Cobbler — Translating from Pascal to Assembler Language

Duffey's proposed Cobbler system [9] uses translation via abstraction and reimplementation in order to compile Pascal [13] programs into PDP-11 assembler language [33]. Cobbler's goal is the creation of extremely efficient object code — code which is comparable in efficiency to the code which could be produced by an expert assembly language programmer. This is a level of efficiency which is beyond any existing compiler and is arguably beyond the abilities of any translator based on transliteration and refinement.

At first glance, it may seem surprising that Cobbler and Satch use the same approach to translation. After all, the problems associated with compiling Pascal do not seem to be very similar to the problems associated with translating Cobol to Hibol. In particular, the goal of the former is efficiency of low level output while the goal of the latter is readability of high level output.

However, the two kinds of translation actually have a great deal in common. Stated generally, the key problem both systems face is that the quality criteria which govern the source are very different from the quality criteria which govern the target. In order to have the freedom to do a good job of satisfying the target criteria, the source must be analyzed and restated in an abstract way which frees it from the constraints of the source criteria.

Example of Cobbler's Compilation

Figs. 16 & 17 (adapted from [9]) show an example of how Cobbler is intended to operate. Fig. 16 shows a Pascal program which initializes a 4x4 array A of bytes to the identity matrix. The program does this a column at a time by setting each column element to zero and then changing the diagonal element to one. Fig. 17 shows the PDP-11 assembler code which would be produced by Cobbler.

```
var   I: 1..4; J: 1..4;
      A: array[1..4, 1..4] of 0..255;
begin
  for J := 1 to 4 do
    begin
      for I := 1 to 4 do A[I,J] := 0;
      A[J,J] := 1
    end
end
```

Fig. 16. The Pascal program INITIALIZE.

```
    MOV   #A,R3
    MOV   #3,R0
L1: MOVB  #1,(R3)+
    CLRB  (R3)+
    CLRB  (R3)+
    CLRB  (R3)+
    CLRB  (R3)+
    DEC   R0
    BGT   L1
    MOVB  #1,(R3)
```

Fig. 17. Cobbler's compilation of Fig. 16.

The code in Fig. 17 is much more efficient than a simple literal translation of Fig. 16 into PDP-11 assembler. The optimizations introduced can be divided into two categories: algorithm independent optimizations and changes to the algorithm.

The algorithm independent optimizations are improvements which any good optimizing compiler might make. The inner loop is unrolled in order to eliminate the overhead engendered by having a loop. The matrix `A` is operated on as a one dimensional vector in order to simplify address calculations. The outer loop is controlled by an auxiliary counter (`R0`) which counts down instead of up. This allows the code to take advantage of the fact that, on the PDP-11, comparison with zero is more efficient than comparison with other numbers. (After each arithmetic operation, condition codes are automatically set which specify whether the result is greater than, equal to, or less than zero.)

For the most part, the optimizations above are straightforward. The first simply involves duplicating the inner loop body, and the second is essentially a strength reduction. However, introducing an auxiliary loop counter is somewhat more complex. If a loop counts from n up to m by s. Then a new loop counter can be introduced which counts from $m-n/s$ down to zero by one. Computation of the old counter is retained so that it can be used within the loop while the new counter is used to control the loop. (In Fig. 17 no trace of this computation remains because the simplification of the addressing calculations has rendered it unnecessary.) The correctness of this transformation is supported by the fact that Pascal prohibits the body of a "`for`" loop from modifying the iteration variable or the bounds of the iteration.

In order to highlight the algorithmic changes introduced by Cobbler, Fig. 18 shows a decompilation of Fig. 17 which undoes the effects of the algorithm independent optimizations discussed above while leaving the algorithmic changes in place. It should be noted that the figure is merely intended as a presentational device. There are a number of reasons why Fig. 18 is not a valid Pascal program. (Most notably, the matrix `A` is declared to have different bounds from those which are presumably associated with other uses of the matrix.)

```
var   I: 1..3; J: 2..5;
      A: array[1..3, 1..6] of 0..255;
begin
  for I := 1 to 3 do
    begin
      A[I,1] := 0;
      for J := 2 to 5 do A[I,J] := 0
    end;
  A[4,1] := 1
end
```

Fig. 18. A decompilation of Fig. 17 into pseudo-Pascal.

Comparison of Fig. 16 with Fig. 18 shows that the computation performed by the target code produced by Cobbler is startlingly different from the computation performed by the source code. In fact, it is probably not appropriate to say that the two pieces of code are using the same algorithm.

Three algorithmic changes have been introduced. The target code avoids redundantly setting the diagonal

elements to zero before setting them to one. The target operates on A in row major order rather than column major order. The target treats A logically as a rectangular 3x5 matrix plus one additional element instead of as a square 4x4 matrix.

Perhaps the most important difference is the switch to row major order. For whatever reason, the programmer chose to use column major order in Fig. 16. This choice clashes with the fact that Pascal stores arrays in row major order. Switching to row major order changes the program so that it references the elements of A in memory storage order. This in turn makes it possible to use auto-increment mode PDP-11 instructions to support the address calculations required.

Undoubtedly the most surprising change is the switch to operating on A as a 3x5 matrix. This makes it much easier to set the appropriate elements of A to one since all these elements are now in the same column.

As will be discussed in the next subsection, Cobbler is able to make the algorithmic changes outlined above because it creates an abstract description of the program which is not constrained by the order of iteration in the loops, or even by the fact that A is declared to be a 4x4 matrix. These changes are arguably beyond the scope of any current optimizing compiler because they require an understanding of what is being computed by the source program as a whole.

If the programmer had written the program as shown in Fig. 18 then any good optimizing compiler could have produced the code in Fig. 17. However, it is implausible that the programmer would have written the program in a form anything like Fig. 18. This is of course partly due to the fact that it is not technically possible to write the program shown in Fig. 18 in Pascal. However, much more importantly, it is not desirable to write programs like Fig. 18. The programmer should not have to worry about detailed efficiency in the source code. Rather, readability should be the primary concern. The source program in Fig. 16 is preferable to the one in Fig. 18 because it is more readable and therefore easier to test, verify, and maintain. (One might argue that Fig. 16 would be even more readable if it operated in row major order. However, the fact that it operates on A as a 4x4 matrix clearly makes the program easier to understand than Fig. 18.)

Design of Cobbler

As shown in Fig. 19, the architecture of Cobbler is similar to the architecture of Satch (see Fig. 12). In particular, the first three stages of abstraction — parsing, plan creation, and grouping — are identical, and are intended to make use of the same modules of KBEmacs. The difference between the lengths of the right hand sides of Figs. 12 & 19 is intended to indicate that creating an efficient PDP-11 implementation of an abstract description is much harder than creating a Hibol implementation.

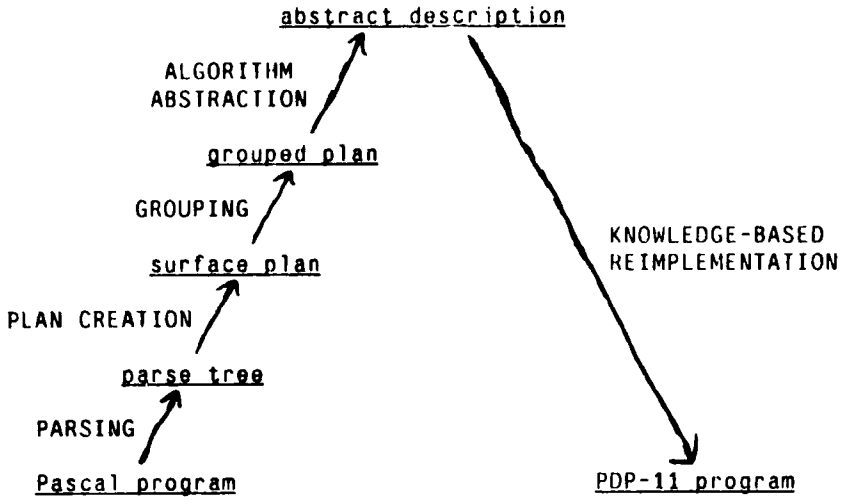


Fig. 19. The architecture of Cobbler.

The final stage of abstraction used by Cobbler (algorithm abstraction) goes beyond the algorithm identification used by Satch. The goal of algorithm abstraction is to identify the various design decisions which were used when writing the Pascal program and then undo them. This leads to a hierarchy of abstract descriptions for the program which are constrained by fewer and fewer design decisions.

When analyzing the program in Fig. 16, the algorithm abstraction module first withdraws the decision to use loops when operating on A. This implicitly withdraws the decision to iterate in column major order as opposed to row major order. It then withdraws the decision to set the diagonal elements to zero before setting them to one. Finally it withdraws the decision to implement A as a Pascal array as opposed to a non-contiguous group of variables. All of these steps could be performed by recognizing standard algorithms in a grouped plan for Fig. 16.

The left side of Fig. 20 summarizes the last step of algorithm abstraction. The 4x4 description represents the net effect of the program in Fig. 16 on the Pascal array A. The abstract description represents the net effect of the program operating directly on the individual matrix elements. The significance of the abstract description is that it gives Cobbler the freedom to consider ways of accessing A other than as a 4x4 array.

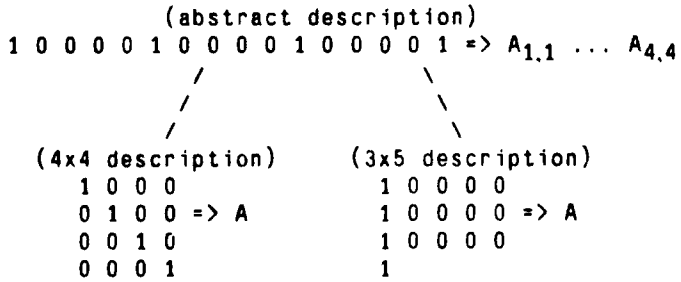


Fig. 20. Some descriptions of Fig. 16 used by Cobbler.

The knowledge-based reimplementation module creates an efficient PDP-11 implementation corresponding to

the abstract description. As one of the first parts of the reimplementation process, Cobbler looks for patterns in the abstract description in order to decide how to use loops in the output program. The recurring pattern "1 0 0 0 0" is discovered. This causes Cobbler to reorganize its understanding of the program into the 3x5 description shown on the right hand side of Fig. 20.

Once the 3x5 description has been created, reimplementation proceeds by investigating a variety of implementation options and then choosing a consistent and efficient set of these options. Following standard Pascal practice, the array A is implemented as a row major order sequence of consecutive bytes in memory. (This decision has to take the other uses of the matrix A into consideration.) Elements of A are addressed by stepping a pointer through memory. Since the inner loop which zeros the non-diagonal elements of A is very small and only iterates four times it is unrolled into a sequence of four separate instructions. Clear-byte instructions are used to zero elements of A.

The key difficulty in making the above design decisions (and the other decisions which are required) is controlling the search process which investigates the various options. Flexibly and efficiently controlling search was the major focus of Duffey's research. He proposed the following approach to the problem.

A data base is used to represent Cobbler's evolving understanding of the implementation. Design decisions are represented in terms of transformations. Each transformation consists of a pattern and a procedural body. Transformations are triggered (causing their bodies to be executed) when their patterns match portions of the data base. The effect of a transformation is to modify the information in the data base, or add new information to the data base.

The key component of the knowledge-based reimplementation module is a *conflict resolution monitor* which controls the triggering of transformations. It exercises control principally by deactivating and activating groups of transformations. Associated with each group of transformations is a function which can create estimates of the costs in time and space associated with the design decision suggested by the group of transformations. (For a discussion of one way in which such estimates can be computed see [14]) The conflict resolution monitor decides which groups of transformations to activate by comparing efficiency estimates.

An important feature of Cobbler is that it does not assume that it will always be able to make an informed choice between the design decisions it is faced with. In order to deal with this problem, Cobbler keeps a record of the design decisions which were used in the source program. In situations where Cobbler is not able to make an informed choice, it uses the relevant source program decision. For example, if no pattern had been found in the abstract description, Cobbler would have used the 4x4 structure suggested by the source program.

It would also be possible for Cobbler to take advice on how to compile a program because Cobbler's processing is based on design decisions which are comprehensible to a programmer.

The discussion above shows how Cobbler is intended to operate. However, Cobbler is not a running system. With the exception of parts of the reimplementation component, no attempt has been made to implement Cobbler.

VI - The Knowledge-Based Translator

Work is currently underway in the Programmer's Apprentice project on the components of a general purpose, knowledge-based translator operating via abstraction and reimplementation. An important virtue of this system is that much of its knowledge of translation will be represented as data rather than procedures. As a result, it will be possible to readily extend the system to cover a wide range of source and target languages.

In order to understand how the knowledge-based translator will operate, it is first necessary to discuss two of the key ideas which underly the Programmer's Apprentice (see [28]). The first idea is the concept of a *cliche*. Programs are not constructed out of arbitrary combinations of primitive programming constructs. Rather, programs are built up by combining standard computational fragments and data structure fragments. These standard fragments are referred to as cliches and form the heart of the Programmer's Apprentice's understanding of programming, just as they form the heart of any person's understanding of programming.

As an example of cliches, consider the Cobol program PAYROLL in Fig. 10. This program contains a number of cliches which can be named and described as follows. The data cliche *keyed-sequential-Cobol-file* specifies how a series of records with keys can be combined into a file. The computational cliche *enumerate-keyed-sequential-Cobol-file* enumerates all of the records in a file taking care of opening and closing the file. The computational cliche *accumulate-keyed-sequential-Cobol-file* writes out a series of records into a file taking care of opening and closing the file. The computational cliche *Cobol-sum* computes the sum of a sequence of numbers.

A crucial feature of cliches is that they can be arranged in a multi-level specialization hierarchy as shown in Fig. 21. The descendants of a cliche in this hierarchy are more specialized cliches which specify how the cliche should be adapted in various specific situations. For example, there is an abstract cliche *enumerate* which has a set of descendants which specify how to enumerate various kinds of data structures (e.g., *enumerate-file* and *enumerate-vector*). Similarly, the middle level cliche enumerate-file has a set of descendants which specify how to enumerate different types of files (e.g., *enumerate-indexed-file* and *enumerate-keyed-sequential-file*). Going one step further, each of these specific file enumeration cliches has a set of descendants which specify exactly what functions are used to open, close, and read files in various different programming language environments (e.g., *enumerate-indexed-Ada-file* and *enumerate-keyed-sequential-Cobol-file*).

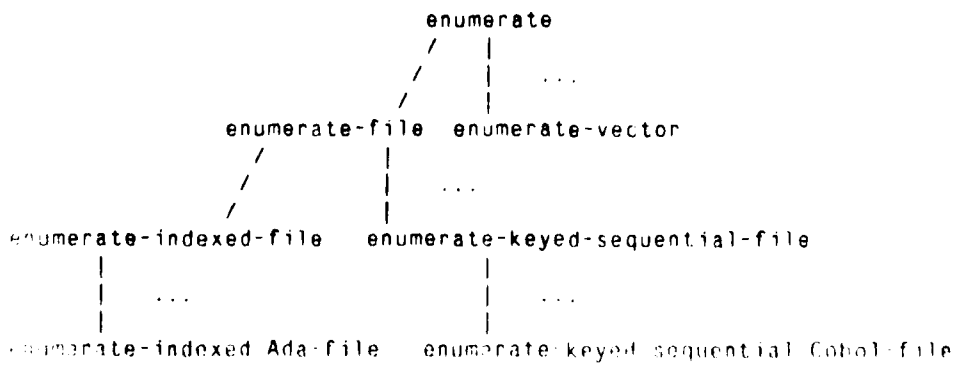


Fig. 21. Examples of specialization relationships between cliches.

A second key idea which underlies the Programmer's Apprentice is the plan representation which was discussed briefly in Section IV. The most important feature of a plan is that it is an abstract representation of a program which captures the key features of the computation while ignoring the syntactic details of particular programming languages. For example, data flow is represented by simple arcs in the plan for a program no matter how it is implemented in the program (e.g., via variables or parameter passing or nesting of expressions).

Both Satch and Cobbler make use of the version of the plan representation which is used by KBEmacs. Since the design of those systems, Rich [17], [18] has developed an extended plan representation called the *plan calculus* which is capable of representing much more information about a program. In particular, the plan calculus is capable of representing data cliches and the specialization relationships between cliches. In contrast, the plan representation used by KBEmacs is only capable of representing computational cliches and only in isolation from each other.

Design of the Knowledge-Based Translator

Fig. 22 shows the way in which plans and cliches can be used as the basis for a knowledge-based translator operating via abstraction and reimplementation. The modules on the left side of the diagram support abstraction. The modules on the right side of the diagram support reimplementation. The key component of the system is a library of cliches like the ones described above. Specialization relationships are used as the basis for the organization of the library.

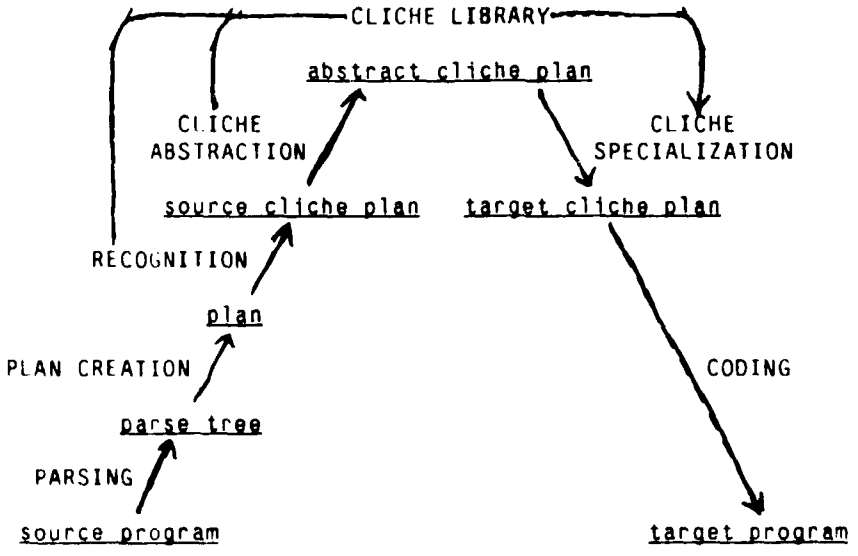


Fig. 22. Translation based on cliches and plans.

The first two steps of abstraction (parsing and plan creation) are exactly the same as in Satch and Cobbler. The last two steps of abstraction (recognition and cliche abstraction) are similar to Cobbler's algorithm abstraction module. A key feature of these modules is that they are data driven — operations based on the cliches stored in the cliche library.

The recognition module scans the grouped plan and determines what source language cliches were used to construct the source program. (This recognition is performed directly on the surface plan and therefore subsumes the grouping performed by Satch and Cobbler.) When applied to the Cobol program in Fig. 10, recognition would reveal that the program was composed of cliches such as keyed-sequential-Cobol-file, enumerate-keyed-sequential-Cobol-file, accumulate-keyed-sequential-Cobol-file, Cobol-count, and Cobol-sum.

The cliche abstraction module creates an abstract plan by replacing specialized plans with the more abstract plans they are specializations of. In the example above, this would yield a plan involving the abstract cliches keyed-sequence, enumerate, accumulate, count, and sum.

The abstract plan attempts not to force any design decisions. It simply states that there are certain sequences of values containing certain data values and keys and that various operations are performed on these values. The point-count feature of the abstract plan is that it is completely neutral between the Cobol program which implements the sequences as files and a Hibol program which implements them as flows or, for that matter, a Lisp program which implements them as lists.

The reimplementation process in Fig. 22 operates in the reverse of the way in which abstraction operates. Cliche specialization selects cliches which specialize the cliches in the abstract plan in a way which is appropriate for the target language. Cliche specialization (which can be looked at as library driven synthesis) is the inverse of cliche abstraction. *However, it is more difficult than* cliche abstraction because it is harder to make design decisions than to discard them.

Coding creates program text corresponding to the specialized cliches which are selected by cliche specialization. Coding is the inverse of parsing, plan creation, and recognition. Inverting recognition and parsing is trivial. However, inverting plan creation is difficult, because information corresponding to the information thrown away by plan creation must be generated. For example, the coding module has to decide how to render data flow aesthetically in the target language using variables and nesting of expressions.

Implementing the Knowledge-Based Translator

Progress has been made toward implementing most of the components in Fig. 22. However, none of these components has yet been completed. Rich and Feldman are currently in the process of implementing the plan calculus together with a general purpose automatic deduction system [19] to support reasoning in it. Extensive work has already been done on designing the library [17].

Given a particular source language, it is not difficult to implement a parsing module. As mentioned above, the plan creation module already exists as part of KBEmacs. This module has to be rewritten so that it operates in the domain of the plan calculus. However, there should be no particular difficulty in doing this.

KBEmacs also contains a coding module analogous to the one needed by the knowledge-based translator. Although there are many improvements which need to be made in this module, it should not be difficult to implement an adequate coding module which operates in the context of the plan calculus.

Work has not been made to implement the cliche specification tool. However, its scope is relatively

should be straightforward. Cliche abstraction is driven by the specialization links in the cliche library. Cliche abstraction is particularly easy because it follows these links in the many-to-one direction.

There has also been no attempt to implement the cliche specialization module. Like cliche abstraction, cliche specialization is driven by the specialization links in the cliche library. However, cliche specialization is harder than cliche abstraction because numerous design decisions have to be made when choosing a path through the specialization links in the one to many direction. It is expected that, like Cobbler, the cliche specialization module will use a variety of estimates and heuristics in order to make design decisions. Also like Cobbler, design decisions detected during cliche abstraction will be used to guide cliche specialization in situations where these heuristics fail to be applicable.

In many ways, the central module in Fig. 22 is the recognition module. Work on this module has been underway for several years. Recognition can be viewed as a parsing task. From this viewpoint, the cliche library is a grammar which can be used to derive plans for programs. In order to determine which cliches were used to construct a given plan one needs to parse the plan. This would be a straightforward task if it were not for the fact that the plan for a program is a graph rather than a string, and cliche instances correspond to subgraphs in the plan rather than substrings.

As a first step toward solving the recognition problem, Brotsky [6] implemented a parser which is able to efficiently parse flow graphs (a restricted form of acyclic directed graph) given a flow graph grammar. Currently, Zelinka [29] is implementing an experimental recognition module which utilizes this graph parser. Further research is required in order to develop effective methods whereby the knowledge-based translator can deal with incomplete recognition.

Once the implementation of the components described above has been completed, it will be possible to use them to construct a general purpose, knowledge-based translator. As mentioned above, a key feature of this system is that it will be data driven with most of its knowledge embedded in the cliche library. Additional research will have to be performed in order to discover how best to represent the heuristics which are an essential part of the specialization component and to a lesser extent of the coder component.

VII - Related Work

There are several areas where active work is in progress on translators. However, essentially all current translators operate via transliteration and refinement. Some translators (e.g., optimizing compilers) do a significant amount of global analysis of the source program. However, it is not clear that any program translator takes the step of attempting to obtain an abstract understanding of the computation being performed by the program as a whole.

Compilers

Compilers are the most common example of translators. They have been well developed over the years and work quite well. As described in text books on compiling (e.g., [1]), the prototypical compiler operates by

transliteration and refinement. The source language is transliterated (via parsing and syntax directed translation) into an intermediate language which is analogous to a machine language. Refinements (optimizations) are then applied to this intermediate representation. Finally, the intermediate language is transliterated into the actual target language. The current developments in compiler research [30] indicate that the basic approach to compilation outlined above is still adhered to.

However, over the years, two trends in compiler research have been moving in the direction of abstraction and reimplementation. One trend is the development of intermediate representations which look more like data flow diagrams and less like particular machine languages. These more abstract representations facilitate the construction of families of compilers which produce output for a variety of target machines. They also facilitate the manipulation of the program when optimizations are being applied. In particular, they makes it easier to keep track of the data flow in a program.

Another trend is toward more powerful optimizations which require a greater understanding of what is going on in a program. Classic peephole optimizations such as locating patterns of instructions for which a special target instruction is available operate in a very local way without any understanding of context. More powerful optimizations such as removing an invariant expression from a loop require a general understanding of the surrounding data flow and control flow. Optimizations such as strength reduction additionally require an understanding of the mathematical properties of the basic operators (e.g., "+" and "*").

The kind of analysis which underlies complex optimizations is a step toward creating an abstract summary of the program being compiled. However, it is only a small step in this direction because the information obtained by analysis is not very abstract. The only abstraction is away from particular data flow and control flow constructs. In addition, the analysis is narrow in scope, aiming only to gather enough information to answer a few specific questions about the program. No attempt is made to obtain a general understanding of the computation performed by the program.

Compiling for Parallel Machines

The problem of compiling a conventional programming language so that it runs efficiently on a parallel machine highlights the strengths and weaknesses of current approaches to optimization. Consider compiling the Fortran program fragment in Fig. 23 for a vector machine. The fragment is a triply nested loop which computes the product of two NxN matrices.

```
      DO 100 J = 1, N
         DO 100 I = 1, N
            DO 100 K = 1, N
               C(I,J) = C(I,J)+A(I,K)*B(K,J)
100   CONTINUE
```

Fig. 23. Loops performing matrix multiplication.

The loops in Fig. 23 can be efficiently executed on a scalar machine. Unfortunately, they cannot be efficiently

executed on the typical vector machine. The problem is that each cycle (after the first) of the innermost loop uses the value computed on the prior cycle leaving little room for vectorization. However, if the loops are interchanged so that the K loop is outermost, then they can be efficiently executed on a vector machine.

The discussion in [2] shows how a compiler for a vector machine can automatically interchange loops in order to improve the efficiency of the code produced. Interchanging two loops changes the order in which computations are performed. Many subcomputations which were performed in the order S1 S2 before the interchange will be performed in the order S2 S1 after the interchange. An interchange is correctness preserving as long as nothing in the original program either requires that S2 follow S1 or prohibits S2 from preceding S1.

A global analysis of the loops in question is a key part of the loop interchange optimization. The compiler must obtain an understanding of the data dependencies between array elements in the loops. This requires an understanding of the data flow involving the arrays (i.e., A, B, and C). It also requires at least a partial understanding of the interaction between the loop iteration variables and the index expressions which select array elements.

In Fig. 23, the index expressions are very easy to understand. However, the index expressions in a loop can be arbitrarily complex. For example, they may be functions of the input data. The analysis of index expressions used by the loop interchange optimization described in [2] is limited to situations where the index expressions are linear functions of the loop iteration variables.

An interesting aspect of loop interchange in particular, and compiler optimizations in general, is that they are deliberately designed to be narrow in scope and independent of whatever computation is being performed. This has the advantage that the various optimizations can be applied in a wide variety of contexts without the need for any special knowledge about the particular algorithms being used. However, it has the disadvantage that the optimizations cannot utilize special knowledge about the particular algorithms being used.

Given the algorithm independent nature of optimizations in general, the level of object code efficiency which can be achieved is very impressive. However, there are definite limits to the efficiency which can be achieved. For example, consider compiling Fig. 23 for a highly parallel machine which has many independent processors. For this kind of machine, optimizations such as loop interchange are not sufficient to produce efficient code. The problem is that for a multiple processor machine, the standard matrix multiplication algorithm is simply the wrong algorithm to use. Special algorithms for matrix multiplication have been developed which are much more efficient when run on a multiple processor machine.

In order to create really good code for a multiple processor machine a compiler would have to recognize that matrix multiplication was being performed in Fig. 23 and then replace the standard algorithm with one of its multiple processor counterparts. The lack of compilers which can make this kind of transformation significantly limits the usability of multiple processor machines. In order to make full use of these machines, programmers have to rewrite their programs in special languages using new algorithms.

Very High Level Language Compilers

A third category of compilers is compilers for general purpose very high level languages. A number of such languages have been designed (e.g., SETL [22], Gist [3], and V [12]). These languages differ from high level languages in that they are more abstract. A good example of this difference is the treatment of data structures. High level languages provide facilities so that the programmer can specify the exact details of how data structures should be implemented. In contrast, very high level languages typically support only a few universal data structures such as sets and mappings. All decisions about how to implement a given set or mapping efficiently are left up to the very high level language compiler. This simplifies what the programmer has to do by removing large parts of the programming task from consideration.

Unfortunately, constructing a compiler for a general purpose very high level language which produces efficient object code has proved very difficult. While these compilers are the subject of active research, it is not clear that such a compiler can be said to exist even in a research setting.

The SETL compiler [11] is implemented more or less along traditional lines with the addition of a special component which selects data structure implementations. However, the key technique which is being pursued as a basis for very high level language compilers is *refinement through transformation* [3], [12]. In this approach a very high level language source program is progressively refined into an efficient target program by applying a sequence of correctness-preserving transformations. The net effect of the transformations is to replace all of the abstract concepts (e.g., set) in the source with concrete concepts (e.g., record or array) in the target. The key problem (which has so far resisted solution) is that there are a vast number of ways in which a source program can be transformed and it is very hard to decide which ones will lead to acceptably efficient results.

Refinement through transformation is basically an example of the transliteration and refinement approach; or rather just refinement. Using transformations has several advantages. In particular, each transformation typically embodies a single implementation decision and is straightforward to understand in isolation. Further, since each transformation is correctness-preserving it is clear that the result produced will be correct.

What is lacking in the transformational approach is a general strategy for making overall design decisions. It is not clear that it is possible to make these decisions on a local basis as individual transformations are applied. One alternate approach would be to pursue all of the major choices, compiling a given program many different ways and then pick the implementation which is best [14]. However, it is not clear that this approach can be practically applied to complex programs where large numbers of choices have to be made.

Another approach which has not yet been tried would be to use abstraction and reimplementation as the basis for choice. The goal would be to recognize patterns of computation in the source program which suggest that particular design choices should be used. A strategy would still be required for selecting between conflicting suggestions. However, this strategy could benefit from having a high level description of the conflict.

Source-to-Source Translation

A number of source-to-source program translators exist. However, as a group, they are not as well developed as compilers and relatively little has appeared in the literature about them. It seems that all current source-to-source translators operate via transliteration and refinement doing relatively little refinement.

Unfortunately, source-to-source translators tend to be incomplete and incorrect. Most of them handle only part (around 90%) of the source language. Further, relatively few source-to-source translators correctly handle the sub-language they are applicable to.

As discussed in Section II, both of these problems stem from difficulties in transliteration. Source language constructs which cannot be reasonably transliterated are not supported. Further, transliteration methods which work most of the time, but not all of the time, are used as if they worked all of the time.

In addition to the problems above, when measured by the criteria of readability, the output of most translators is not particularly good. Although serviceable, the output produced seldom comes anywhere near the goal of being what the programmers would have written had they been writing in the target language.

Due to the difficulties above, it is not accurate to refer to typical source-to-source translation systems as *automatic* systems. It is more accurate to describe them as *human-assisted* translation systems. In order to obtain correct (let alone aesthetic) output, human intervention is usually required. The user has to edit the source program (to remove untranslatable constructs) and/or the target program (to correct errors and improve the translation).

As a straightforward example of a translator, consider the Lisp 1.6 to Interlisp translator implemented by Samet [21]. This translator operates purely by transliteration. It does no refinement. Although reasonably efficient output is produced, the translator makes no attempt to create aesthetic output. In particular, there is no attempt to create Interlisp-style output. Rather, a set of functions is defined in Interlisp which, as much as possible, allows Interlisp to simulate Lisp 1.6. For example, instead of translating the source program into Interlisp syntax, the Interlisp reader is modified so that it can read in a program in Lisp 1.6 syntax. In [21], Samet identifies a number of features of Lisp 1.6 which his translator cannot handle. The user is required to edit the source program in order to eliminate these features. Samet also describes several features of Lisp 1.6 which are translated in ways which are often, but not always, correct. The translation produced has to be carefully tested in order to check that these over-simple transliterations have not led to any problems.

At first glance, it might appear that translation between two dialects of Lisp should be easy. However, this is not the case. In fact, Lisp supports a number of features which are spectacularly difficult to translate. For example, a Lisp program can construct a new Lisp program and then execute this new program. Consider a Lisp 1.6 program which constructs a Lisp 1.6 program and then calls it as a subroutine. The program would have to be translated into an Interlisp program which constructs an Interlisp program. It is very unlikely that this kind of translation could be performed without using abstraction and reimplementation of the most powerful kind.

Another straightforward translator is the Fortran to Lisp translator implemented by Putman [16]. Like Samet's translator, this translator operates purely by transliteration, doing no refinement. The translator produces

readable output. However, it deliberately attempts to create Fortran-style output as opposed to Lisp-style output. The translation is supported by a set of functions which allow Lisp to simulate the Fortran runtime environment. This approach introduces a significant overhead which causes a translated program to run several times slower than the Fortran source program. Pitman's translator is far superior to Samet's translator in that, except for one or two very obscure features, all of the features of Fortran are translated correctly all of the time.

A third translator in this vein is the Fortran to Jovial translator implemented by Boxer [4]. Like the translators above, it operates purely by transliteration. The output of the translator is not intended for human consumption and no attempt is made to make it particularly readable or to render it in Jovial-style. (The examples in [4] indicate that the output is similar in style to the Ada shown in Fig. 3.) The translator only handles a subset of Fortran. It succeeds in translating from 90% to 100% of the typical input module. User intervention is required to complete the translation.

The Lisp to Fortran translator developed by Boyle [5] is interesting because it is based on the transformational approach discussed in the last subsection. The translator handles an applicative subset of Lisp which does not include such hard to translate features as the ability to create and execute new Lisp code. Readability is not a goal of the translation. Rather, readability of the output is abandoned in favor of producing reasonably efficient Fortran code. As discussed in [5], this translator is perhaps best thought of as a compiler of Lisp into Fortran rather than a source-to-source translator.

Boyle's translator operates by transliterating the Lisp source into an extension of Fortran and then transforming this extended Fortran into ordinary Fortran. The transformation process is controlled by dividing it into a number of phases. Each phase applies transformations selected from a small set. The transformations within each set are chosen so that conflicts between transformations will not arise.

Boyle's translator is successful not because it has solved the problems faced by very high level language compilers, but rather because it succeeds in avoiding them. First, compared to SETL, Gist, and V, Lisp is not very abstract. Therefore there are fewer complex design decisions which have to be made. Second, the design decisions are small enough in number that it is possible to find a fixed set of choices which works reasonably well for all of the Lisp programs being translated. These fixed choices are embedded in the translator through the choice of phases and transformations. Lists are always implemented the same way. Recursion is always simulated in the same way. This leads to the production of Fortran programs which are reasonably efficient, but typically far from optimally efficient.

Commercially Available Source-to-Source Translators

In addition to the in-house translation systems described above, a number of translators are commercially available. One area where several translators are available is translating between assembler languages for various microprocessors. The discussion in [25] compares three commercial available translators between 8080 assembler and 8086 assembler. An in-house attempt at a translator between Z80 assembler and MC6809 assembler is discussed in [23]. All four translators operate primarily by transliteration on a instruction by instruction basis

and do little or no refinement. They all operate on only a subset of the source language and use simplistic transliterations which are not correct in all contexts. Human intervention is often required in order to obtain correct output. The translations produced are also quite inefficient, consisting of from 3 to 6 times as many instructions as the source. One of the 8080 to 8086 translators (XLT86 from Digital Research Inc.) uses global data and control flow analysis in order to guide the choice of transliteration for instructions. It produces output which is significantly more efficient and more often correct than the other translators.

Another area where a number of translators are available is translating between various languages used for business data processing (e.g., Cobol, RPGII, and PL/I). Numerous translators exist (for example, see [31], [32]). Substantive information about the internal operation of these translators is hard to obtain, however several things are clear from their external descriptions. They do not handle the whole source language. In general, they only succeed in translating 90% to 95% of typical source programs. They do not always produce correct output. (In [32], the user is specifically instructed to test and debug the translations produced.) Examples suggest that the output is not particularly readable, and that the output was probably created primarily through transliteration.

Code Restructuring

An interesting subcategory of source-to-source translators is systems which translate a program from a language back into the same language. The goal of these systems is to create output which is more readable than the input. In particular, these systems typically seek to render unstructured source programs in a structured form. Given that the source and target languages are the same, it is a relatively straightforward matter to make sure that the entire source language is handled correctly. However, it is far from straightforward to produce output which really is significantly more readable than the input. Many of these systems are little more than pretty printers and are of marginal use. However, at least one system (Recoder [7]) is a true translator and creates highly structured output.

Recoder operates on Cobol programs in three stages. The first stage creates a flow chart-like graph representing the source program. The key feature of the graph representation is that all control flow is represented by explicit arcs which are independent of the Cobol constructs which were originally used to implement the control flow. The second stage applies correctness-preserving transformations to the graph in order to rearrange the graph into a structured form. The third stage creates a new Cobol program based on the rearranged graph.

Recoder represents a step toward the abstraction and reimplementation approach because the abstraction which it uses is clearly the driving force behind the translation. However, the step is strictly limited by a number of factors. The graph representation used is not very abstract. The only abstraction is away from particular control flow constructs. No attempt is made to recognize the algorithms being used in the source program or to abstract away from them.

Natural Language Translation

An interesting area which is closely related to program translation is natural language translation. Work on natural language translation started by using transliteration and, in a quest for high quality output, is now moving in the direction of translation via abstraction and reimplementation.

Almost all of the natural language translation systems which are in actual regular use today operate via transliteration and refinement (see [26]). In general, these systems produce output which is very rough, but which is readable to a person who is familiar with the subject area. A good example of such a system is the Paho system [26] which translates from Spanish to English.

Paho operates by transliterating the source text on a sentence by sentence basis. This transliteration is carried out for the most part on a word by word basis with a small amount of inter-word analysis to take care of issues such as providing correct translations for idioms, and rearranging the adjectives in a noun phrase. (Adjectives follow nouns in Spanish whereas they precede nouns in English). The practicality of this kind of transliteration depends heavily on a number of convenient correspondences between the basic structure of Spanish and English (e.g., the near identicality of word order, and the fact that Spanish pronouns are more heavily marked for gender than English pronouns).

Paho is not capable of refining the English it produces. Manual post-editing is required in order to generate an acceptable translation. The biggest weaknesses of Paho is that it knows very little about syntax and nothing about the meaning of the sentences being translated. Further, it has no knowledge of interactions between sentences.

In the quest for higher quality translations than the ones generated by systems like Paho, translators are now being developed which operate more in the vein of abstraction and reimplementation. A good example of such a translator is the Eurotra system [15] which is currently being developed to translate between the major western European languages. Eurotra uses semantically annotated syntactic parse trees as an abstract representation for the sentences being translated. Analysis (abstraction) and synthesis (reimplementation) components convert source languages into parse trees and parse trees into target languages respectively.

Eurotra is not a true abstraction and reimplementation system because the annotated parse trees are not independent of the source and target languages. Procedural *transfer components* are required in order to convert a source language specific parse tree into a target language parse tree.

It is expected that Eurotra will produce significantly better output than Paho. However, it is expected that Eurotra will still fall short of high quality translation. In particular, although Eurotra has much more syntactic understanding than Paho, its semantic and inter-sentential understanding is still quite weak.

In order to achieve high quality translation, natural language translation systems have to be able to obtain an in-depth understanding of the text being translated. One approach to this is the recent work on *knowledge-based machine translation* (see [8]). This work has succeeded in demonstrating natural language translation via abstraction and reimplementation. The abstract description used by this approach is a language independent representation of the conceptual dependencies in the text. Knowledge-based machine translation is intended to operate by first analyzing the entire source text in order to determine its meaning and then recapturing this

meaning in the target language using the syntactic structure of the source as a guide for what to say when.

Although knowledge-based machine translation holds the promise of generating very high quality output, more work has to be done before a translator following this approach will be practical. In particular, as with translation via abstraction and reimplementation in general, there is a significant problem with incompleteness. Considerable further research has to be done before it will be possible to achieve anywhere near a complete understanding of arbitrary passages of source text. However, perfection is not required. Human translators are unable to translate technical texts unless they understand the technical area being discussed.

Acknowledgments

Under the supervision of the author, G. Faust designed and implemented the Satch system and R. Duffey designed the Cobbler system. C. Rich (with the assistance of Y. Feldman) and L. Zelinka (building on the earlier work of D. Brotsky) are laying the foundations for a generalized translation system based on abstraction and reimplementation. C. Rich made a number of suggestions which greatly improved both the form and content of this paper. Discussions with D. Chapman, R. Duffey, G. Faust, H. Reubenstein, G. Parker, and L. Zelinka helped clarify the ideas presented. Special thanks are due C. Ciro for her assistance with the illustrations.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, "Compilers - Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986.

[2] J.R. Allen and K. Kennedy, "Automatic Loop Interchange", Proc. of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices V19 #6, June 1984.

[3] R. Balzer, "Transformational Implementation: An Example", IEEE TSE V7 #1, January 1981.

[4] R.K. Boxer, "A Translator From Structured Fortran to Jovial/J73", Proc. of the IEEE National Aerospace and Electronics Conference (NAECON-83), 1983

[5] J.M Boyle and M.N. Muralidharan, "Program Reusability through Program Transformation", IEEE Trans. on Software Engineering V10 #5, September 1984.

[6] D.C. Brotsky, "An Algorithm for Parsing Flow Graphs", (MS Thesis) MIT/AI/TR-704, March 1984.

[7] E. Bush, "The Automatic Restructuring of Cobol", Proc. of the IEEE Conf. on Software Maintenance, November 1985.

[8] J. Carbonell, R. Cullingford and A. Gershman, "Knowledge-based machine translation", IEEE Trans. on Pattern Analysis and Machine Intelligence V3 #4, 1981.

[9] R.D. Duffey II, "Formalizing the Expertise of the Assembler Language Programmer", MIT/AI/WP-203, September 1980.

[10] G.G. Faust, "Semiautomatic Translation of Cobol into Hibol", (MS Thesis) MIT/LCS/TR-256, March 1981.

[11] S.M. Freudenberger, J.T. Schwartz and M. Sharir, "Experience with the SETL Optimizer", ACM TOPLAS V5 #1, January 1983.

[12] C. Green *et. al.*, "Research on Knowledge-Based Programming and Algorithm Design -- 1981", Kestrel Institute, Palo Alto CA, 1981.

[13] K. Jensen and N. Wirth, "Pascal User Manual and Report", Springer-Verlag, 1975.

[14] E. Kant, "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", (PhD thesis) Stanford AIM-338, September 1979.

[15] M. King, "Eurotra: A European system for machine translation", ISSCO, Univ. of Geneva, Switzerland, 1980.
[16] K.M. Pitman, "A Fortran -> Lisp Translator", Proc. of the 1979 Macsyma Users' Conference, June 1979.
[17] C. Rich, "Inspection Methods in Programming", (PhD thesis) MIT/AI/TR-604, June 1981.
[18] —, "A Formal Representation for Plans in the Programmer's Apprentice", Proceedings IJCAI-81, August 1981.
[19] —, "The Layered Architecture of a System for Reasoning about Programs", Proceedings IJCAI-85, August 1985.
[20] G.G. Ruth, S. Alter and W.A. Martin, "A Very High Level Language for Business Data Processing", MIT/LCS/TR-254, 1981.
[21] H. Samet, "Experience with Software Conversion", Software — Practice and experience V11 #10, 1981.
[22] J.T. Schwartz, "On Programming", An Interim Report on the SETL Project, Courant Institute of Mathematical Sciences, New York University, June 1975.
[23] M.F. Smith and B.E. Luff, "Automatic Assembler Source Translation from the Z80 to the MC6809", IEEE Micro V4 #2, April 1984.
[24] G.L. Steele Jr., "Common Lisp: the Language", Digital Press, Maynard MA, 1984.
[25] R. Taylor and P. Lemmons, "Upward Migration Part I: Translators", Byte V7 #4, June 1983.
[26] A.B. Tucker Jr., "A Perspective on Machine Translation: Theory and Practice", CACM V27 #4, April 1984.
[27] R.C. Waters, "A Method for Analyzing Loop Programs", IEEE TSE V5 #3, May 1979.
[28] —, "The Programmer's Apprentice: A Session with KBEmacs", IEEE TSE V11 #11, November 1985.
[29] L.M. Zelinka, "Automated Program Recognition", MIT/AI/WP-279, December 1985.
[30] Proc. of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN notices V19 #4, June 1984.
[31] Brochures describing various translation products, Cap Gemini Dasd Inc., NY NY, 1985.
[32] Brochures describing various translation products, Dataware Inc., Orchard Park NY, 1985.
[33] "PDP 11/34 Processor Handbook", Digital Equipment Corp., Maynard MA, 1976.
[34] "Fortran IV Language", publication C28-6515-6, IBM, White Plains NY, 1966.
[35] "Scientific Subroutine Package Version III Programmer's Manual", publication GH20-0205-4, IBM, White Plains NY, 1970.
[36] "American National Standard Cobol", publication GC28-6396-5, IBM, White Plains NY, 1973.
[37] "Military Standard Ada Programming Language", ANSI/MIL-STD-1815A, U.S. Department of Defense, January 1983.

END

DATE

FILMED

JAN

1988